UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS

Lucas Finger Roman

GEOMETRICKS: UM FRAMEWORK DE ESTRUTURAS DE DADOS GEOMÉTRICAS EM C++

FLORIANÓPOLIS
2020

LUCAS FINGER ROMAN

Geometricks: Um framework de estruturas de dados geométricas em C++

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina, como requisito necessário para obtenção do grau de Bacharel em Ciências da Computação

Florianópolis, 1 de dezembro de 2020

UNIVERSIDADE FEDERAL DE SANTA CATARINA

LUCAS FINGER ROMAN

Esta Monografia foi julgada adequada para a obtenção do título de Bacharel em Ciências da Computação, sendo aprovada em sua forma final pela banca examinadora:

Orientador: Prof. Dr. Maicon Rafael Zatelli
Universidade Federal de Santa Catarina - UFSC

Prof. Dr. Alvaro Junio Pereira Franco
Universidade Federal de Santa Catarina - UFSC

Profa. Dra. Jerusa Marchi
Universidade Federal de Santa Catarina - UFSC

Florianópolis, 1 de dezembro de 2020

Este trabalho é dedicado a todas as pessoas que me ajudaram nesta longa jornada, desde os meus grandes professores de ensino infantil até todas as pessoas que me apoiaram para que isso fosse possível.

Agradecimentos

Primeiramente gostaria de agradecer aos meus pais Humberto e Leslie por toda a educação e carinho que recebi durante toda minha vida. Pessoas incríveis que me ensinaram todos os valores que levo para a vida toda, e que amo do fundo do meu coração. Também gostaria de agradecer à Nani, que cuidou de mim como uma segunda mãe desde que eu era pequeno e ao filho dela, Bruno, que considero como um irmão.

Gostaria de agradecer aos meus avós, com os quais tive uma relação muito próxima e que, infelizmente, não estão mais aqui para ler este trabalho por todo o tempo que passamos juntos. Também gostaria de agradecer aos meus tios e primos, por todos os natais em família que passamos juntos, apesar da distância. Gostaria de agradecer aos meus amigos, reais e virtuais, que me deram todo o apoio emocional para a realização deste trabalho e sempre estiveram comigo nos momentos bons e nos momentos ruins, seja para dar risada ou para desabafar. Agradeço às minhas terapeuta e psiquiatra pela ajuda psicológica com este trabalho e com minhas situações do dia a dia. Agradeço também ao professor Maicon, que me ajudou profundamente com a realização deste trabalho, e à banca examinadora Jerusa e Álvaro, por tomarem tempo para o ler e avaliar.

Por último, gostaria de agradecer a todos os professores com os quais tive o prazer de ter aulas, desde o ensino infantil até o ensino superior. Nunca teria chegado tão longe sem o apoio e aprendizado que tive com todos vocês e espero poder botar em uso todo o conhecimento acumulado na minha vida profissional como forma de retribuir.

Resumo

Objetos geométricos, tais como pontos, retas, polígonos e cubos, não possuem uma ordem intrínseca para estruturas de dados clássicas, devido as suas diversas dimensões serem independentes. Uma possível solução para o armazenamento de dados multidimensionais seria a multi-indexação dos dados por cada uma de suas dimensões. Porém, buscas que, ou utilizem características geométricas dos dados, ou usem mais de uma das diversas dimensões ao mesmo tempo, ainda têm seu desempenho degradada, junto da necessidade de manter diversas cópias atualizadas. Dito isto, como solução para armazenamento e otimização de algoritmos que utilizem estes objetos, estruturas de dados geométricas, tais como *rtree*, *quadtree*, *kdtree* e *octree*, que particionam o espaço geométrico de busca são utilizadas. Este trabalho visa a implementação, na linguagem C++, de uma biblioteca para as estruturas *quadtree* e *kdtree*, com foco na alta personalização dos dados e documentação das técnicas de implementação utilizadas de forma didática, com o intuito de ajudar outras pessoas a implementar estruturas similares de maneira eficiente e genérica.

Palavras-chave: Estruturas de dados geométricas. *Framework*. C++. *Quadtree*. *Octree*. *KDTree*. *Grid*. Algoritmos.

Abstract

Geometric objects, such as points, lines, polygons and cubes, are not intrinsically ordered in respect to classic data structures, due to their multiple dimensions being independent. A possible solution to such problem would be to index the data separately by each of its dimensions. With that said, such solution fails to address either queries that look up multiple dimensions at the same time or queries that are dependent on geometric features of the data, while also having to maintain multiple synchronized copies of the data. In order to efficiently store and optimize algorithms that may use such objects, geometric data structures, such as rtrees, quadtrees, kdtrees and octrees, which subdivide the geometric space, are used. This work implements a library, in the C++ programming language, for the quadtree and kdtree structures, focusing on user customization and documentation in a didactic way, so that it may help other people to implement likewise data structures in a generic and efficient way.

Keywords: Geometric data structures. Framework. C++. Quadtree. Octree. KDTree. Grid. Algorithms.

Sumário

1 Introdução

Diferentes problemas computacionais requerem diferentes algoritmos, que processam os dados, e diferentes estruturas de dados, que organizam os dados para uma resolução eficiente. Além disso, diferentes tipos de problemas agrupam-se em diferentes categorias. Uma destas categorias é a de problemas geométricos, que para uma resolução eficiente requer estruturas de dados geométricas.

Problemas geométricos envolvem dados geométricos, tais como pontos, retas e polígonos. Estruturas de dados geométricas, desta forma, se aproveitam de características pertinentes a estes dados para organizar o espaço, diminuindo o espaço de busca, o gasto de memória ou aumentando o desempenho. Técnicas comuns envolvem o agrupamento de dados logicamente próximos, simplificação de regiões em outras menores e repartição do espaço em elementos geométricos específicos. Pode-se citar aqui, por exemplo

1. para o armazenamento de uma imagem binária, usualmente se requer pelo menos um bit por pixel, em que o valor 1 representa a presença de um objeto e o valor 0 representa a ausência. Para redução do custo de memória, pode-se aproveitar a característica de píxeis próximos usualmente terem um valor em comum para definir uma estrutura que simplifica grandes regiões da imagem com o valor 1 para apenas um descritor da região. Regiões com o valor 0 podem ser omitidas, visto que a imagem é binária e apenas 2 valores são possíveis. Desta forma, o consumo pode ser reduzido drasticamente.

2. uma das possíveis otimizações para busca de vizinho mais próximo em bancos de dados é agrupar dados próximos em uma estrutura que preserve logicamente sua proximidade. Desta forma, não é necessário procurar o banco inteiro em consultas de vizinho mais próximo. Isso traz um ótimo ganho de desempenho, visto que não apenas bancos de dados armazenam alta quantidade de dados, assim como também requer acessos à memória secundária, que é ordens de magnitude mais lento que acesso à memória principal.

Para agilizar o desenvolvimento de projetos que necessitam destas estruturas, dados e algoritmos, bibliotecas genéricas são desenvolvidas em diferentes linguagens. Desta forma, cada projeto tem a opção de apenas importar uma destas bibliotecas, sem a necessidade de criar uma própria. Uma destas linguagens é a linguagem C++, que permite abstração de dado e maior controle do hardware ao mesmo tempo. Diferentes bibliotecas têm diferentes objetivos. Algumas bibliotecas, como a [FLANN - Fast Library for Approximate Nearest Neighbors], focam em desempenho e paralelização de algoritmos geométricos. Outras

bibliotecas, como a [Boost.Geometry], focam em usabilidade, extensibilidade e elementos de programação genérica. Este trabalho foca na implementação de um *framework* de estruturas de dados geométricas em C++ com um foco mais didático e de boa usabilidade.

1.1 Objetivo

1.1.1 Objetivo Geral

Implementação e documentação de um *framework* de estruturas de dados geométricos em C++ de forma didática.

1.1.2 Objetivos Específicos

- Levantar algoritmos e estruturas de dados geométricos.
- Estudar as estruturas e algoritmos a implementar.
- Implementar as estruturas e algoritmos em C++.
- Permitir às estruturas personalização para que estas aceitem tipos de dados e métricas de distância definidos pelo usuário.
- Documentar as implementações de forma didática, exemplificando seu uso e sua complexidade computacional.

1.2 Organização

O presente trabalho está organizado da seguinte forma:

- O capítulo 2 contém a fundamentação teórica a respeito das estruturas de dados geométricas implementadas para este trabalho.
- O capítulo 3 contém uma comparação entre o *framework* desenvolvido com outros *frameworks* para estruturas de dados geométricas.
- O capítulo 4 contém uma referência da API, descrevendo os métodos implementados, a motivação para a convenção de chamadas da API e técnicas de implementação utilizadas no desenvolvimento.
- O capítulo 5 contém discussões sobre a implementação da API e os resultados obtidos.
- O capítulo 6 contém a conclusão e trabalhos futuros.

2 Fundamentação Teórica

Para o desenvolvimento do *framework*, são selecionadas as estruturas Árvore K-D, *QuadTree*, *OcTree*, *RTree* e *Grid*, devido a suas importâncias. Para estas estruturas, são testadas diversas técnicas de implementação, abordadas no capítulo 4. Desta forma, este capítulo não se preocupa em detalhar e aprofundar a implementação das estruturas em si, contendo apenas um breve resumo e descrição algorítmica das operações pertinentes a elas.

2.1 *Grid*

A estrutura de dados *grid* é o exemplo mais simples de estrutura geométrica. Dado um espaço 2D, é possível dividir este espaço em subespaços de tamanhos iguais, de forma que cada posição (x, y) mapeie para um único *grid*. Desta forma, dado *grids* de tamanho $n \times m$ e um plano de tamanho $p \times q$, pode-se armazenar uma matriz de tamanho $\lceil \frac{p}{n} \rceil \times \lceil \frac{q}{m} \rceil$, em que cada célula mapeia elementos para um *grid* específico e cada elemento (x, y) é mapeado para o *grid* presente na posição $(\lceil \frac{x}{n} \rceil, \lceil \frac{y}{m} \rceil)$ da matriz.

Visto sua simplicidade, *grids* apresentam operações de inserção, remoção, construção e atualização eficientes. Outras operações, como testes de colisão, vizinho mais próximo e busca por segmento têm sua eficiência dependente da densidade dos dados. Caso os dados sejam esparsos, essas operações são eficientes. Caso contrário, o desempenho degrada para o desempenho da estrutura interna aos *grids*, comumente implementados como listas encadeadas, com um custo adicional de memória para armazenar os *grids* vazios.

Devido a sua simplicidade e alto desempenho, *grids* são comumente utilizados como estruturas geométricas para cenas esparsas.

2.1.1 Construção

Uma estrutura de *grid* pode ser construída tanto estática como dinamicamente. Pode-se definir um *grid* como uma matriz de subestruturas, inicializando todas essas estruturas junto do *grid* ou como uma matriz de ponteiros para diferentes subestruturas, inicializando essas subestruturas dinamicamente. No caso do *grid* conter poucos elementos ou esses elementos não serem completamente esparsos, inicializar o *grid* dinamicamente diminui o custo de memória do programa. Para uma matriz A de i linhas e j colunas, esta operação possui sua complexidade computacional equivalente a complexidade de inicializar a estrutura interna a cada *grid*.

2.1.2 Inserção de elemento

Para inserir um objeto em um *grid*, primeiro é feito um mapeamento deste para células específicas da matriz. Caso um objeto pertença logicamente a mais de uma célula, ele é inserido em todas as células a qual ele pertence. Isso, porém, pode causar um aumento no consumo de memória e degradação de desempenho no caso do objeto ser muito grande. Algumas técnicas podem então ser citadas como maneiras de diminuir o consumo de memória.

- Guardar os objetos em um arranjo externo ao *grid* e inserir nos quadrantes apenas índices para um arranjo intermediário de elementos, o qual contém índices para os objetos em si.
- Guardar informações extra nos bits mais significativos dos índices, evitando maior uso de memória.

Caso os elementos internos sejam guardados com uma *free list*, a complexidade de inserção é O(1).

2.1.3 Remoção de elemento

Um elemento pode ser removido buscando as células do *grid* que o contém e removendo todos os elementos que apontam para o elemento em si presentes nessas células. Essa operação depende da estrutura de dados usada para guardar os elementos dentro de cada célula, podendo tomar até O(n) caso uma simples lista seja usada para guardar os elementos e o *grid* seja muito denso.

2.2 Árvores K-D

Árvores K-D [Bentley 1975] são árvores binárias K dimensionais que permitem rápida busca de dados, separando o espaço em K diferentes planos para diminuir o número de testes necessários. Cada um dos níveis da árvore divide os dados de entrada com um hiperplano associado a uma das dimensões, separando o espaço em 2 subespaços. Dados menores que o hiperplano vão para um subespaço, enquanto dados maiores vão para o outro. Esses nós intermediários podem optar por guardar parte dos dados ou somente o valor do hiperplano que divide o espaço.

Uma árvore K-D possui algoritmos de construção, busca por N vizinhos mais próximos e busca por segmento eficientes. Porém, inserção de elementos de forma aleatória degrada o desempenho da árvore, devido à sua grande dificuldade de balanceamento.

Árvores K-D podem ser utilizadas para guardar objetos estáticos da cena. Desta forma, testes de colisão com estes objetos são realizados eficientemente, enquanto outra

Algoritmo 1: ConstroiArvoreKD

Entrada: Sequência de elementos S, Dimensão D, Número de Dimensões dim
Saída: Árvore K D balanceada

```
1  início
2      Menores ← ∅
3      Maiores ← ∅
4      Pivo ← SelecionaPivo(S, D)
5      para cada elemento ∈ S faça
6          se elemento[D] < Pivo[D] então
7              Menores ← Menores + elemento
8          senão
9              Maiores ← Maiores + elemento
10         fim
11     fim
12     se |Menores| > 0 então
13         Esquerda ← ConstroiArvoreKD(Menores, (D + 1)  mod  dim)
14     senão
15         Esquerda ← ∅
16     fim
17     se |Maiores| > 0 então
18         Direita ← ConstroiArvoreKD(Maiores, (D + 1)  mod  dim)
19     senão
20         Direita ← ∅
21     fim
22     retorna Nodo(Esquerda, Direita, D, Pivo)
23 fim
```

estrutura de dados que implementa métodos de inserção se encarrega de objetos dinâmicos, devido a constante necessidade de remoção e inserção.

2.2.1 Construção

Uma árvore K-D pode ser construída conforme o algoritmo 1. Nas linhas 2 e 3, são criados 2 conjuntos de elementos a serem processados nos níveis inferiores da árvore. Um pivô é selecionado na linha 4, onde o pivô normalmente é dado pela mediana dos conjuntos de entrada na dimensão *dim*. As linhas 5 a 11 separam o resto dos elementos nos 2 conjuntos definidos anteriormente, com os elementos maiores que o pivô sendo processados no filho à direita e os elementos menores que o pivô sendo processados no filho à esquerda. As linhas 12 a 16 realizam as chamadas recursivas caso tenha algo a processar para o filho à esquerda. As linhas 17 a 21 fazem a mesma coisa para o filho à direita. Por fim, retorna-se o nó da árvore raiz. Um nodo pode ser representado pelo conjunto `(Esquerda,Direita,Dimensão,Objeto)`.

A complexidade deste algoritmo é dada pela fórmula 2.1.

$$T(n) \leftarrow T(\frac{n}{2}) + T(\frac{n}{2}) + O(SelecionaPivo(S, D)) + O(n) \tag{2.1}$$

Pode-se perceber algumas otimizações quanto a este algoritmo. Para decidir o pivô e separar as duas sequências para as chamadas recursivas ao mesmo tempo, o algoritmo *quickselect* [Hoare 1961] tem complexidade O(n) [Blum et al. 1973], juntando as linhas 4 a 11 do algoritmo 1 em apenas uma chamada ao *quickselect* sem a necessidade de espaço adicional. Esta chamada é feita para os dados de entrada na construção nível a nível, com o pivô selecionado sendo o objeto a ser armazenado no nível atual, os elementos a esquerda do pivô são processados no filho à esquerda e os elementos a direita do pivô no filho à direita. Como outras alternativas, pode-se preprocessar e manter k cópias ou índices ordenadas dos dados externamente em cada dimensão, selecionando o valor encontrado na mediana do vetor D ao processar a dimensão D e mantendo o cuidado para não reutilizar índices. É possível também ordenar os dados pela dimensão atual dentro de cada nível, porém a complexidade do algoritmo é pior que a do *quickselect* e também é possível aplicar um algoritmo para achar a mediana em um subconjunto dos pontos de tamanho definido, não garantindo balanceamento perfeito em troca de melhor desempenho na construção. Para cada uma dessas alternativas, observa-se as seguintes relações de complexidade

- Quickselect

$$T(n) \leftarrow T(\frac{n}{2}) + T(\frac{n}{2}) + O(n) \tag{2.2}$$

- K cópias externas ordenadas em diferentes dimensões

$$\begin{aligned} C(n) &\leftarrow C(\frac{n}{2}) + C(\frac{n}{2}) + O(n) \\ T(n) &\leftarrow O(k \times n \log n) + C(n) \end{aligned} \tag{2.3}$$

- Ordenação de cada subnível

$$T(n) \leftarrow T(\frac{n}{2}) + T(\frac{n}{2}) + O(n) + O(n \log n) \tag{2.4}$$

- Mediana em subconjunto

$$T(n) \leftarrow T(\frac{n}{2}) + T(\frac{n}{2}) + O(n) \tag{2.5}$$

Resolvendo as equações acima pelo teorema mestre [Bentley, Haken e Saxe 1980], obtém-se

- Quickselect O(n log n)
- K cópias externas ordenadas em diferentes dimensões O(k $\times$ n log n)
- Ordenação de cada subnível O(n $\log^2$n)

- Mediana em subconjunto O(n log n)

Dito isto, uma possível comparação entre calcular a mediana em um subconjunto e utilizar o *quickselect* é de que, embora ambos os casos precisem de O(n) operações, para o caso do subconjunto o valor de n pode ser definido, evitando uma busca por todos os pontos de entrada. Isto pode diminuir drasticamente o valor de n, acelerando o processo de construção da árvore em troco de deixá-la levemente desbalanceada.

Outra possível observação é no armazenamento dos dados. Uma escolha a ser feita é o posicionamento final dos elementos. A árvore pode ser construída tanto como um arranjo, com a vantagem de localidade de cache, em que a raiz se encontra no centro do arranjo e cada um dos nós filhos se encontra na metade da distância dos extremos até o nó atual, tanto como uma árvore tradicional, com nós sendo alocados de forma aleatória na memória. No caso de alocação aleatória, um alocador de memória pode alocar n blocos de antemão, diminuindo a complexidade de futuras alocações para nós da árvore a O(1). Para esta implementação, foi decidido utilizar a árvore em arranjo. A seção 4.3.6 explora mais a fundo a representação e implementação dessa estrutura em arranjo.

2.2.2 Busca vizinho mais próximo

Uma operação muito importante é, dado um conjunto de dados C e um ponto externo P, achar um ponto $X \in C$ que contenha a distância $D = f(P, X)$ mínima. Como motivação para esta operação, são citadas as seguintes aplicações [Roussopoulos, Kelley e Vincent 1995]:

- Um usuário, ao clicar num ponto qualquer da tela, tem uma busca pelos 5 objetos mais próximos ao ponto clicado, sendo útil para selecionar objetos num sistema de GUI.
- Num sistema de astrofísica, achar uma estrela próxima a um ponto clicado pelo usuário envolve no caso ingênuo múltiplas comparações não sucedidas.
- Algoritmos de vizinho mais próximo podem ser usados com diferentes métricas de distância em sistemas de recomendação para usuários [Sarwar et al. 2001].

2.2.2.1 Um vizinho

A ideia de busca de vizinho mais próximo em Árvores K-D se aproveita da divisão dos níveis por um hiperplano para descartar ramos de busca. Em um determinado nível, observa-se a posição do ponto atual em relação ao hiperplano. Se a menor distância obtida for menor que a distância do ponto ao hiperplano, é impossível um ponto presente do outro lado ter uma distância menor, possibilitando o descarte de todo o outro ramo. Observa-se

que não se pode fazer isso com o plano em que o ponto de busca se encontra, sendo sempre necessário recursar a este lado. Dito isto, outro ponto a ser observado é que a menor distância é atualizada durante a recursão, o que permite a realização do teste após a chamada desta com o novo valor. Inicializa-se o teste pela raiz, com uma distância mínima infinita e testando a dimensão 0, descendo pela árvore comparando o ponto de busca com o ponto atual. Caso este ponto seja maior na dimensão atual que o ponto atual, recursa-se à direita. Caso contrário, à esquerda. A cada etapa da recursão, aumenta-se a dimensão atual em 1 $\mod dim$, onde dim é o número de dimensões do dado. Ao chegar nos nós folhas então, atualiza-se a distância e a recursão retorna. Desta forma, o algoritmo pode ser descrito pelo Algoritmo 2, com a inicialização deste sendo Raiz = raiz da Árvore K-D, Dim = 0 e D = ∞. Este algoritmo possui complexidade O($\log n$) caso o número de dimensões k

Algoritmo 2: VizinhoMaisProximo

Entrada: Nó Raiz, Ponto P1, Dimensao Dim, Distancia D, Função de distância Distancia, Número de Dimensões dim

Saída: Ponto P2 mais próximo a P1

```
1  início
2      se P1[Dim] < Raiz[Dim] então
3          Lado ← Esquerda
4      senão
5          Lado ← Direita
6      fim
7      se Lado == Esquerda então
8          P2 ← VizinhoMaisProximo(Raiz.Esquerda, P1, (Dim + 1) mod dim, D)
9          se Distancia(Raiz.P, P1) < D então
10             D ← Distancia(Raiz.P, P1)
11             P2 ← Raiz.P
12         fim
13         se Distancia(P1, Raiz.HiperPlano < D) então
14             P2 ← VizinhoMaisProximo(Raiz.Direita, P1, (Dim + 1) mod dim,
                   D)
15         fim
16     senão
17         P2 ← VizinhoMaisProximo(Raiz.Direita, P1, (Dim + 1) mod dim, D)
18         se Distancia(Raiz.P, P1) < D então
19             D ← Distancia(Raiz.P, P1)
20             P2 ← Raiz.P
21         fim
22         se Distancia(P1, Raiz.HiperPlano) < D então
23             P2 ← VizinhoMaisProximo(Raiz.Esquerda, P1, (Dim + 1) mod dim,
                   D)
24         fim
25     fim
26     retorna P2
27 fim
```

seja baixo. Porém, caso o número de dimensões K seja muito grande, o desempenho pode degradar para complexidade linear.

2.2.2.2 Estendendo o algoritmo para K vizinhos

O algoritmo pode ser facilmente estendido se observado que, para as chamadas recursivas, não se deve somente comparar a distância do nó atual ao hiperplano, mas também levar em consideração o número de pontos coletados. Caso o número de pontos coletados seja menor que o número desejado, o outro ramo é obrigatoriamente observado. Caso contrário, verifica-se a distância ao hiperplano com a pior distância entre os nós saída. Na hora de atualização então deve-se sempre retirar o ponto de saída com a maior das distâncias ao encontrar um candidato melhor. O passo a passo é descrito no algoritmo 3, com sua inicialização sendo Raiz = raiz da Árvore K-D, Dim = 0 e C = $\emptyset$. O funcionamento deste algoritmo é muito similar ao algoritmo 2, com a única diferença sendo em quando descartar os ramos recursivos e o tipo de retorno da função.

2.2.3 Busca por segmento

Uma busca por segmento permite ao usuário pesquisar por todos os elementos que se encontram dentro de um determinado conjunto de valores (min, max). Por exemplo, dado uma árvore KD contendo diversos pontos 2D, uma possível pesquisa é procurar por todos os pontos que se encontram dentro do intervalo $[(x = 0, y = 0), (x = 100, y = 100)]$.

Para esta busca, pode-se observar que todos os pontos à esquerda do nó atual possuem valor menor que o elemento na dimensão. Desta forma, para realizar uma busca, caso o valor mínimo do segmento na dimensão atual se encontre à direita do nó atual, o ramo do filho da esquerda pode ser completamente descartado, visto que todos os elementos à esquerda são menores que o mínimo e estão fora do alcance de busca. O mesmo efeito pode ser observado com o valor máximo e o filho a direita. Outra observação possível é que as outras dimensões do elemento atual só precisam ser comparadas com o resto do segmento de entrada no caso do valor na dimensão se encontrar entre $[min(dimensao), max(dimensao)]$. Neste caso, o algoritmo deve recursar aos filhos da esquerda e direita. Representa-se o algoritmo em pseudo-código conforme o algoritmo 4, onde dimensões é o número de dimensões do dado.

2.3 *Quadtree*

Quadtrees [Samet 1984] são uma estrutura de dados hierárquica bidimensional bastante utilizada em algoritmos de computação gráfica, processamento de imagem, robótica e jogos. Sua implementação consiste em dividir uma região retangular em subsequentes 4 regiões de tamanhos iguais, com cada uma destas sub-regiões se dividindo recursivamente

Algoritmo 3: KVizinhoMaisProximo

```
   Entrada: Nó Raiz, Ponto P1, Dimensao Dim, K vizinhos, Coleção C, Função de Distância Distancia,
            Número de Dimensões dim
   Saída: Coleção contendo K pares de ponto e distância C
 1 início
 2     se P1[Dim] < Raiz[Dim] então
 3         Lado ← Esquerda
 4     senão
 5         Lado ← Direita
 6     fim
 7     se Lado == Esquerda então
 8         C ← KVizinhoMaisProximo(Raiz.Esquerda, P1, (Dim + 1) mod dim, K, C)
 9         se C.qtdElementos < K então
10             D ← Distancia(Raiz.P, P1)
11             P2 ← Raiz.P
12             C.insere([D, P2])
13         senão
14             se Distancia(Raiz.P, P1) < C.maior então
15                 D ← Distancia(Raiz.P, P1)
16                 P2 ← Raiz.P
17                 C.remove(Ponto com maior distância a P1)
18                 C.insere([D, P2])
19             fim
20         fim
21         se Distancia(P1, Raiz.HiperPlano) < C.maior ou C.qtdElementos < K então
22             C ← KVizinhoMaisProximo(Raiz.Direita, P1, (Dim + 1) mod dim, K, C)
23         fim
24     senão
25         C ← KVizinhoMaisProximo(Raiz.Direita, P1, (Dim + 1) mod dim, K, C)
26         se C.qtdElementos < K então
27             D ← Distancia(Raiz.P, P1)
28             P2 ← Raiz.P
29             C.insere([D, P2])
30         senão
31             se Distancia(Raiz.P, P1) < C.maior então
32                 D ← Distancia(Raiz.P, P1)
33                 P2 ← Raiz.P
34                 C.remove(Ponto com maior distância a P1)
35                 C.insere([D, P2])
36             fim
37         fim
38         se Distancia(P1, Raiz.HiperPlano < C.maior) ou C.qtdElementos < K então
39             C ← KVizinhoMaisProximo(Raiz.Esquerda, P1, (Dim + 1) mod dim, K, C)
40         fim
41     fim
42     retorna C
43 fim
```

em níveis menores e armazenando pontos, regiões ou outros objetos geométricos em cada um de seus nós.

Devido a sua divisão hierárquica do espaço, *quadtrees* otimizam operações de busca ou união de regiões espaciais. Além disso, diferentes tipos de *quadtree* armazenam seus dados de maneiras diferentes. Enquanto uma *quadtree* de região ou de detecção de objeto divide uma imagem em diversos blocos até se encontrar um bloco uniforme, onde todos os valores presentes sejam o mesmo, outros tipos de *quadtree* podem armazenar objetos geométricos de cena e armazenar o mesmo objeto em diferentes sub blocos, acelerando operações como detecção de colisão [Ericson 2004].

Uma *quadtree* possui eficientes algoritmos de busca, busca por segmento, inserção,

Algoritmo 4: BuscaSegmento

Entrada: Nó Raiz, Ponto Min, Ponto Max, Dimensão D, Número de Dimensões dim
Saída: Conjunto C contendo todos os pontos dentro do segmento

```
1  início
2      C ← ∅
3      se Raiz.Elemento[D] < Min[D] então
4          se Raiz.Direita então
5              retorna BuscaSegmento(Raiz.Direita, Min, Max, D + 1 mod dim)
6          fim
7      senão
8          se Max[D] < Raiz.Elemento[D] então
9              se Raiz.Esquerda então
10                 retorna BuscaSegmento(Raiz.Esquerda, Min, Max, D + 1 mod dim)
11             fim
12         senão
13             se Min < Raiz.Elemento < Max então
14                 C ← C + Raiz.Elemento
15             fim
16             se Raiz.Esquerda então
17                 C ← C ∪ BuscaSegmento(Raiz.Esquerda, Min, Max, D + 1 mod dim)
18             fim
19             se Raiz.Direita então
20                 C ← C ∪ BuscaSegmento(Raiz.Direita, Min, Max, D + 1 mod dim)
21             fim
22         fim
23     fim
24     retorna C
25 fim
```

remoção, *bulk loading*, sendo uma ótima opção para objetos dinâmicos. Também devido a sua capacidade de atribuir um valor a uma área, são ótimas opções para armazenar imagens binárias onde grande porções apresentam o mesmo valor.

O *framework* implementa uma *quadtree* que armazena objetos geométricos. As seções a seguir tratam de algoritmos utilizados para esta implementação.

2.3.1 Construção

Para a construção de uma *quadtree*, é definido um *bounding rect* que deve conter todos os objetos inseridos na árvore. Uma *quadtree* pode ser construída com *bulk loading* calculando o *bounding rect* que engloba todos os objetos de entrada e inserindo elemento a elemento conforme a seção 2.3.2. Desta forma, dois diferentes algoritmos de construção podem ser definidos: o algoritmo 5 e o algoritmo 6.

Um *bounding rect* é definido pelos valores (xmin, xmax, ymin, ymax) e um nó da árvore é definido pelos valores (retângulo,eFolha,índice), onde xmin representa o valor mais à esquerda do *bounding rect*, xmax o valor mais à direita do *bounding rect*, ymin o valor mais baixo do *bounding rect*, ymax o valor mais alto do *bounding rect*, retângulo o *bounding rect* armazenado pelo nó, eFolha representa se o nó é folha e índice representa o índice para o primeiro filho ou para o primeiro elemento.

Algoritmo 5: ConstroiQuadTree

Entrada: *Bounding rect* Retangulo
Saída: *Quadtree*

```
1 início
2 |  retorna Nodo(Retangulo,Folha,Vazio)
3 fim
```

Algoritmo 6: ConstroiQuadTree

Entrada: ConjuntoDeObjetos C
Saída: *Quadtree*

```
1  início
2  |  Retangulo ← (Max, Min, Max, Min)
3  |  para cada elemento ∈ C faça
4  |  |  rect ← BoundingRect(elemento)
5  |  |  se rect.xmin < Retangulo.xmin então
6  |  |  |  Retangulo.xmin ← rect.xmin
7  |  |  fim
8  |  |  se rect.xmax > Retangulo.xmax então
9  |  |  |  Retangulo.xmax ← rect.xmax
10 |  |  fim
11 |  |  se rect.ymin < Retangulo.ymin então
12 |  |  |  Retangulo.ymin ← rect.ymin
13 |  |  fim
14 |  |  se rect.ymax > Retangulo.ymax então
15 |  |  |  Retangulo.ymax ← rect.ymax
16 |  |  fim
17 |  fim
18 |  Raiz ← Nodo(Nodo(Retangulo, Folha, Vazio))
19 |  para cada elemento ∈ C faça
20 |  |  Insere(Raiz, elemento)
21 |  fim
22 |  retorna Raiz
23 fim
```

2.3.2 Inserção

O algoritmo de inserção em uma *quadtree* depende do tipo de *quadtree* implementado. Para este trabalho, a estrutura escolhida para implementação contém os objetos apenas nos nós folha, com cada objeto inserido aparecendo uma vez para cada nó folha que o contém. Desta forma, pode ocorrer redundância de dados.

O algoritmo é definido então desta maneira:

1. Caso o *bounding rect* da *quadtree* não contenha o objeto, a inserção não pode ser realizada.

2. Se o *bounding rect* contém o objeto e o nó não é um nó folha, é realizada uma chamada recursiva deste algoritmo para os 4 filhos do nó atual.

3. Se o nó atual é um nó folha e a inserção do elemento faz com que a capacidade do nó ultrapasse o máximo, ocorre uma divisão do nó em 4 nós folhas, o nó atual passa a ser intermediário e é realizada uma chamada recursiva para os 4 filhos.

4. Caso a inserção do objeto não ultrapasse o limite de objetos no nó folha, o objeto é adicionado ao nó.

Um possível problema a ser observado neste algoritmo é, no caso de vários objetos serem adicionados na mesma região, um nó decide se dividir infinitamente, criando um laço infinito. Outro possível problema a ser observado é o fato do *bounding rect* de alguns nós ser muito pequeno para se dividir. Ambos estes problemas podem ser resolvidos limitando a quantidade de vezes que um nó pode se subdividir. Isso pode ser feito controlando um tamanho mínimo para os *bounding rect* e uma profundidade máxima para a árvore. Desta forma, um pseudocódigo é observado no algoritmo 7. As linhas 2 a 4 contém uma verificação se o elemento está contido na árvore. As linhas 5 a 11 mostram o caso de um nó não folha, apenas chamando uma recursão aos níveis mais a baixo. As linhas 12 a 19 mostram o caso da árvore se subdividir, cuidando dos casos especiais para a árvore não se subdividir infinitamente ou ter retângulos não capazes de se dividir mostrados acima. As linhas 20 a 22 apenas adicionam um elemento a um nó folha. Este algoritmo é inicialmente chamado com o valor p = 0. A profundidade máxima da árvore e o tamanho mínimo do retângulo são definidos como metadados na criação da árvore.

2.3.3 Remoção

Como uma observação para a remoção de elementos, observa-se que o espaço da *quadtree* sempre será dividido da mesma forma. Desta forma, é possível apenas realizar uma busca na árvore e remover as referências aos objetos nos nós folhas. Porém, uma outra possível observação é que a remoção de elementos pode possibilitar a junção de nós adjacentes em um nó não folha, caso isso faça com que as propriedades de limite de objeto e níveis da árvore continuem a ser respeitadas. Esse processo é aplicado recursivamente dos níveis mais baixos da árvore para cima, juntando nós não utilizados em outros nós e salvando memória.

Dito isto, uma remoção em uma *quadtree* pode ser simplificada conforme o algoritmo 8. As linhas 2 a 4 verificam se o objeto está contido no retângulo. As linhas 16 a 24 tratam um nó folha: é feito uma busca nos elementos contidos pelo nó folha pelo elemento a ser removido. Caso ele seja encontrado, ele é removido da lista de elementos daquele nó e um valor de verdadeiro é retornado para sinalizar ao nó pai talvez a necessidade de junção de

Algoritmo 7: InsereQuadTree

```
   Entrada: Quadtree arvore, Objeto obj, Retângulo rect, Profundidade p
   Saída: Booleano que informa se a inserção foi feita com sucesso
 1 início
 2     se arvore.retangulo.NaoContem(rect) então
 3         retorna Falso
 4     fim
 5     se arvore.NaoEFolha() então
 6         InsereQuadTree(arvore.TopoEsquerda, obj, rect, p + 1)
 7         InsereQuadTree(arvore.TopoDireita, obj, rect, p + 1)
 8         InsereQuadTree(arvore.BaixoEsquerda, obj, rect, p + 1)
 9         InsereQuadTree(arvore.BaixoDireita, obj, rect, p + 1)
10         retorna Verdadeiro
11     senão
12         se p < arvore.ProfundidadeMaxima e arvore.NumElementos = arvore.Max e
             arvore.retangulo.tamanho não for muito pequeno então
13             Subdivide(arvore)
14             InsereQuadTree(arvore.TopoEsquerda, obj, rect, p + 1)
15             InsereQuadTree(arvore.TopoDireita, obj, rect, p + 1)
16             InsereQuadTree(arvore.BaixoEsquerda, obj, rect, p + 1)
17             InsereQuadTree(arvore.BaixoDireita, obj, rect, p + 1)
18             retorna Verdadeiro
19         fim
20         arvore.elementos.adiciona(obj, rect)
21         arvore.NumElementos ← arvore.NumElementos + 1
22         retorna Verdadeiro
23     fim
24 fim
```

nó. As linhas 5 a 15 fazem chamadas recursivas aos nós filhos. Caso um dos nós sinalize a necessidade de junção das folhas, o nó verifica com os filhos adjacentes para uma possível junção e sinaliza o nó pai caso ela aconteça.

Uma outra possível escolha a ser feita é de apenas realizar a junção de nós caso o nó folha fique vazio após a remoção. Isso pode acabar custando um pouco mais de memória à árvore, mas melhora o desempenho no caso médio, devido a menor quantidade de operações necessárias.

2.3.4 Busca

Para realizar uma busca na árvore, pega-se o *bounding rect* do objeto e são feitas chamadas recursivas assim como na inserção. Porém, ao encontrar o objeto no primeiro nó filho, não é necessário continuar buscando, apenas retornando uma referência ao objeto. Isso é descrito brevemente nos seguintes passos:

Algoritmo 8: RemoveQuadTree

Entrada: Quadtree arvore, Objeto obj, Retângulo rect
Saída: Booleano que informa se a remoção foi feita com sucesso e se é necessário juntar os nós

```
1  início
2      se arvore.retangulo.NaoContem(rect) então
3          retorna Falso
4      fim
5      se arvore.NaoEFolha() então
6          Necessario ← Falso
7          para cada filho F de arvore faça
8              se RemoveQuadTree(F, obj, rect) então
9                  Necessario ← Verdadeiro
10             fim
11         fim
12         se Necessario então
13             Necessario ← TentaJuntarNos()
14         fim
15         retorna Necessario
16     senão
17         para cada elemento E contido por arvore faça
18             se E é igual a obj então
19                 Remove(E)
20                 retorna Verdadeiro
21             fim
22         fim
23         retorna Falso
24     fim
25 fim
```

1. Se o nó não for um nó folha, a busca é chamada recursivamente em ordem para os nós filhos. Caso uma das buscas seja bem sucedida, retorna o objeto.

2. Se o nó for um nó folha, testa-se cada um dos elementos contidos pelo nó com o elemento de busca. Caso o elemento seja encontrado, retorna o elemento.

Este processo é demonstrado no algoritmo 9. As linhas 2 a 4 verificam se o retângulo contém o elemento. As linhas 5 a 12 buscam cada um dos nós filhos do nó atual e retornam o primeiro valor encontrado. As linhas 14 a 19 testam os valores encontrados nos nós folhas com o valor passado como argumento.

2.3.5 Busca por segmento

Uma busca por segmento é muito semelhante a uma busca por objetos, com a diferença de que não temos um objeto específico a ser buscado, e sim uma área de

Algoritmo 9: BuscaQuadTree

```
   Entrada: Quadtree arvore, Objeto obj, Retângulo rect
   Saída: Elemento ou valor vazio
 1 início
 2     se arvore.NaoIntersecta(rect) então
 3         retorna Vazio
 4     fim
 5     se arvore.NaoEFolha() então
 6         para cada filho F de arvore faça
 7             Resultado ← BuscaQuadTree(F, obj, rect)
 8             se Resultado != Vazio então
 9                 retorna Resultado
10             fim
11         fim
12         retorna Vazio
13     senão
14         para cada elemento E de arvore faça
15             se E é igual a obj então
16                 retorna E
17             fim
18         fim
19         retorna Vazio
20     fim
21 fim
```

busca. Dessa forma, o algoritmo 10 se assemelha muito com o algoritmo 9, tendo sua única diferença nas linhas 13 a 17. Quando um objeto é encontrado, ele não é retornado automaticamente, e sim adicionado a um conjunto de retorno. Esses conjuntos têm sua união geral retornada pelo algoritmo.

2.4 *Octree*

Octrees [Meagher 1982] são estruturas de dados de indexação espacial muito semelhares à *quadtrees*. Enquanto *quadtrees* trabalham em 2 dimensões e se subdividem em 4 filhos por nível intermediário, estruturas de mais dimensões se subdividem em 2^D filhos, onde D é o número de dimensões. Uma *octree* é uma dessas estruturas, dividindo um espaço de 3 dimensões em 8, o que dá o nome à estrutura.

Devido à grande semelhança entre *quadtrees* e *octrees*, essa seção aponta as diferenças entre as duas, não descrevendo a fundo os algoritmos.

Além disso, as duas se diferem um pouco na aplicação, com *octrees* sendo mais utilizadas na área de jogos para otimização de operações como detecção de colisão para objetos 3D. Como outra aplicação cita-se *ray-tracing* [Whang et al. 1995].

Algoritmo 10: BuscaSegmentoQuadTree

```
Entrada: Quadtree arvore, Retângulo rect
Saída: Conjunto de elementos
1  início
2      se arvore.NaoIntersecta(rect) então
3          retorna Vazio
4      fim
5      se arvore.NaoEFolha() então
6          Conjunto saida ← ∅
7          para cada Filho f de arvore faça
8              saida ← saida ∪ BuscaSegmentoQuadTree(f, rect)
9          fim
10         retorna saida
11     senão
12         Conjunto saida ← ∅
13         para cada Elemento e de arvore faça
14             se rect.contem(e.rect) então
15                 saida ← saida ∪ e
16             fim
17         fim
18         retorna saida
19     fim
20 fim
```

Como diferença, aponta-se a organização dos dados. Enquanto *quadtrees* trabalham com retângulos, *octrees* dividem o espaço em cubos. Dessa forma, onde os algoritmos de *quadtree* falam em retângulos, pode-se traçar um paralelo a algoritmos de *octree*, apenas trocando o tipo geométrico e o número de divisões do espaço.

3 Trabalhos relacionados

Neste capítulo são apresentadas breves descrições de *frameworks* similares para estruturas de dados geométricas. Feitas as devidas descrições, descreve-se um paralelo entre esses *frameworks* e este trabalho.

3.1 *Frameworks* relacionados

3.1.1 Boost.Geometry

[Boost.Geometry] é uma biblioteca em C++ sob a licença [Boost Software License 2003], desenvolvida para resolução de problemas geométricos de forma genérica, permitindo fácil extensão para tipos de dado definidos pelo usuário. Apesar de ter um foco maior em problemas geográficos, o *framework* pode ser utilizado para outros problemas geométricos, tais como jogos, computação gráfica, *widgets*, robótica, astronomia, entre outros. Dito isto, o *framework* usa de programação genérica, *tag dispatching*, *meta functions*, *concepts*, *trait classes* e macros, de forma que o usuário registre tipos de dados e os plugue com a biblioteca em si.

A biblioteca é dividida em diferentes módulos, cada módulo responsável por uma função geométrica específica. Dentre estes módulos, observa-se o módulo de indexação, o módulo de adaptadores, o módulo de algoritmos, o módulo de aritmética, o módulo de *concepts*, o módulo de coordenadas, o módulo de metaprogramação e o módulo de objetos geométricos. Nas seguintes subseções são exploradas algumas características pertinentes a cada um destes módulos.

3.1.1.1 Módulo de indexação

O módulo de indexação é responsável por organizar os dados, permitindo buscas por vizinho mais próximo de diferentes estruturas geométricas, por região, por intersecção e por predicados definidos pelo usuário. Além disso, é permitido a junção de diferentes tipos de busca através de uma linguagem de domínio e o uso de iteradores de busca, de forma que, para pegar o próximo resultado, basta incrementar o iterador. Para este módulo, apenas a estrutura de dados *RTree* é implementada, com planos de implementação de outras estruturas para o futuro. Para a estrutura *RTree*, são definidas as heurísticas de balanceamento *linear*, *quadratic*, *rstar* e *pack*.

3.1.1.2 Módulo de adaptadores

O módulo de adaptadores possibilita tipos de dados externos à biblioteca a serem usados em algoritmos e estruturas da biblioteca. Este módulo já vem com algumas implementações para dados da *stdlib* do C++ e para arranjos.

3.1.1.3 Módulo de algoritmos

O módulo de algoritmos implementa diferentes algoritmos usados com objetos geométricos. Usando estruturas de dados adaptadas ao *framework* ou nativas do *framework*, pode-se calcular diferentes tipos de distância entre diferentes objetos, área de diferentes objetos, perímetro, intersecção entre outros. Este módulo também disponibiliza iteradores.

3.1.1.4 Módulo de aritmética

O módulo de aritmética traz operações básicas de álgebra para trabalhar com pontos e vetores.

3.1.1.5 Módulo de *concepts*

O módulo de *concepts* é usado pela própria biblioteca em tempo de compilação para diferenciar implementações e garantir propriedades.

3.1.1.6 Módulo de coordenadas

A biblioteca permite o uso de diversos tipos de coordenadas para trabalhar com os dados. Um dado pode se adequar bem a coordenada cartesiana, mas devido ao seu foco em extensibilidade e programação genérica, são disponibilizados outros tipos de coordenadas, tais como coordenadas geográficas ou coordenadas esféricas.

3.1.1.7 Módulo de metaprogramação

Contém funções que usam do sistema de *template* da linguagem C++ em tempo de compilação para diversos efeitos que resultam em código de melhor desempenho ou mais genérico.

3.1.1.8 Módulo de objetos geométricos

Contém a definição dos diferentes tipos de dado usados dentro da biblioteca. Como tipos de dados definidos pela biblioteca, pode-se citar ponto, ponto 2D, linha, polígono, multiponto, multilinha, multipolígono, caixa, anel e segmentos.

3.1.2 FLANN - *Fast Library for Approximate Nearest Neighbors*

[FLANN - Fast Library for Approximate Nearest Neighbors] é uma biblioteca desenvolvida em C++ para busca de vizinho mais próximo de alto desempenho sob a licença [BSD License]. A biblioteca possui um sistema de escolher automaticamente o melhor algoritmo e os melhores parâmetros de busca de acordo com o conjunto de dados passado como parâmetro. Além de C++, a linguagem possui *bindings* para as linguagens C, Matlab, Python e Ruby.

Esse *framework* se preocupa em operações de alto desempenho, não sendo genérico mas possuindo implementações em GPU e *multithread*.

Para a indexação, são implementados os métodos *Locality Sensitive Hashing*, *K means*, árvore KD e *linear*.

A arquitetura do *framework* é orientada a objetos, com uma classe abstrata que define o método de *nearest neighbor* enquanto diferentes classes concretas o implementam com algoritmos diferentes, usando o padrão de projeto de *template* [Gamma et al. 1994].

3.1.3 GameplayKit

[Apple GameplayKit] é um *framework* orientado a objetos implementado em Swift e Objective-C cujo principal foco é auxiliar no desenvolvimento de jogos. Visto a necessidade de jogos armazenarem objetos presentes no jogo em uma estrutura de acesso rápido e eficiente, devido a alta quantidade de objetos na cena e nos requisitos de jogos serem processados em tempo real para dar ilusão de movimento, o *framework* disponibiliza estruturas geométricas como *Quadtree*, *Octree* e *RTree*.

Além de possuir implementações das 3 estruturas geométricas mencionadas acima, o *framework* ainda conta com um sistema de entidades para definir os objetos do jogo, além de outros módulos usados para desenvolver jogos, tais como inteligência artificial, máquina de estados, física, áudio, geração de números aleatórios e primitivas geométricas (obstáculo, esfera, polígono e círculo) que compõe o mundo e são usadas pelo sistema de física.

As estruturas de dados geométricas disponibilizadas pelo *framework* necessitam receber como entrada, além dos elementos a serem inseridos, a posição em pontos de cada um dos elementos.

3.1.4 Boxtree

[Boxtree] é uma biblioteca que, dado a localização de um ponto em duas ou três dimensões, os organiza em uma *Quadtree* ou *Octree* eficientemente usando [OpenCL].

3.2 Comparativos

Feita uma análise de *frameworks* relacionados, pode-se fazer um comparativo destes *frameworks* com o desenvolvido neste projeto. Como critério de avaliação, aponta-se 6 diferentes tópicos:

- Linguagem de programação, abrangendo deste a linguagem de programação utilizada na implementação até *bindings* para outras linguagens.
- Tipo de arquitetura. Se usa orientação a objetos ou programação genérica.
- Estruturas suportadas. Quais estruturas são implementadas pelo *framework*.
- Usabilidade. Que tipos de operações o usuário precisa implementar para integração de seus dados.
- Desempenho. Se a biblioteca é focada no máximo de desempenho ou não.
- Área de uso. Onde se espera que o *framework* seja usado.

Framework	Linguagem	Arquitetura	Estruturas	Usabilidade	Desempenho	Área
Boost.Geometry	C++	Programação genérica	*RTree*	Especialização de *traits*	Médio	Geográfica
FLANN	C++, C, Matlab, Python, Ruby	Orientado a objeto	*KDTree*. Implementa outros algoritmos, mas não estruturas	Carregar dados dentro de matrizes definidas pela biblioteca	Alto desempenho. GPU	Buscas de *nearest neighbor*
Apple Gameplay Toolkit	Swift, Objective-C	Programação orientada a objetos e entidades	*RTree*, *Quadtree*, *Octree*	Componentes criados a partir de agentes. Passa o ponto de localização do objeto para estruturas	Alto	Jogos
Boxtree	Python	?	*Quadtree*, *Octree*	?	Alto	?
Geometricks	C++	Programação genérica	*KDTree*, *Quadtree*	Especialização de *traits*	Objetivo secundário	Didático

4 Desenvolvimento

Este capítulo contém técnicas de implementação utilizadas pelo *framework*, assim como detalhes quanto a arquitetura e usabilidade do *framework*.

A seção 4.1 introduz a arquitetura geral do *framework*, com uma breve descrição de cada um dos módulos implementados. A seção 4.2 exemplifica a utilização do *framework*, com exemplos de uso dos módulos e um dado de usuário a ser personalizado. Por fim, a seção 4.3 trata de técnicas de implementação utilizadas pela biblioteca e de decisões de implementação tomadas para cada uma das estruturas desenvolvidas.

O código fonte pode ser encontrado em <https://github.com/mitthy/TCC>.

4.1 Arquitetura

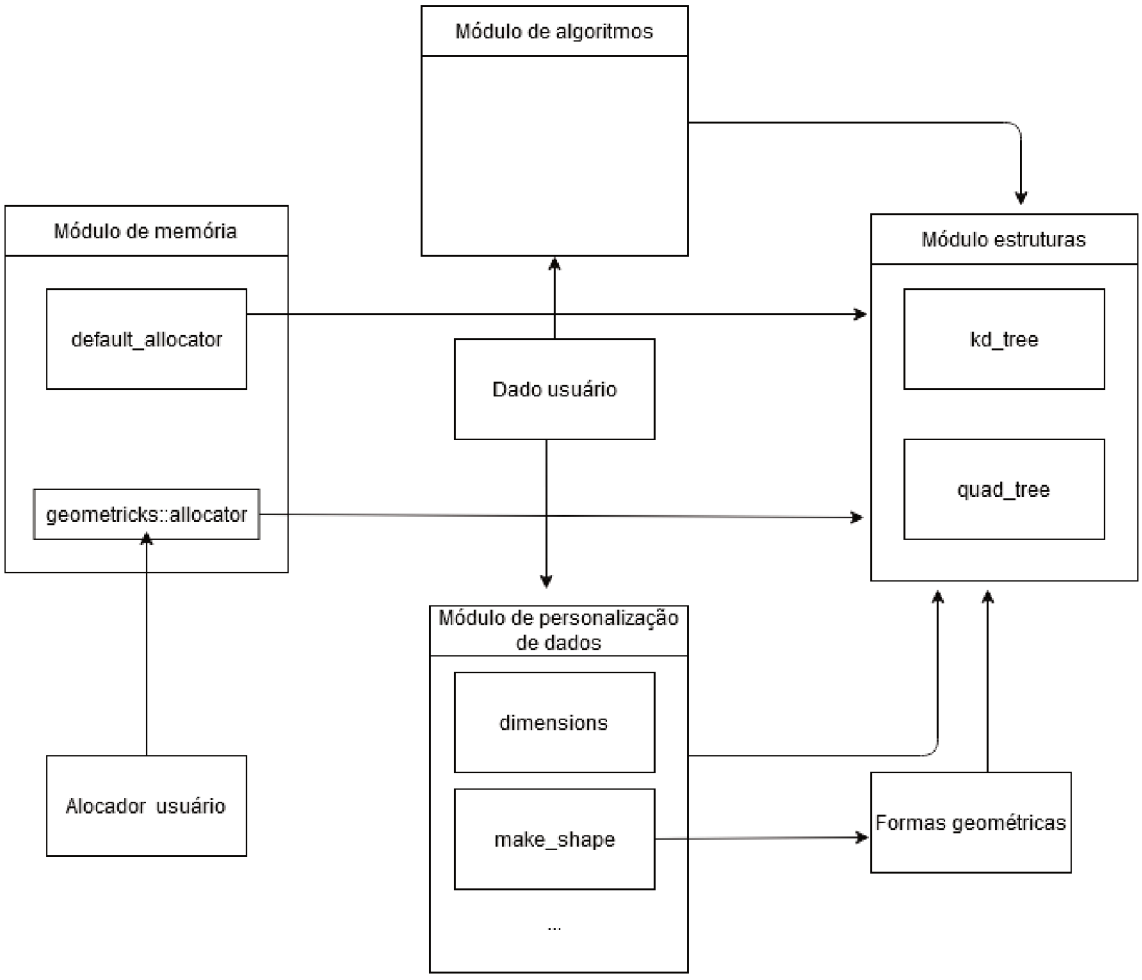

Figura 1 – Visão geral da arquitetura do *framework*.

Para a arquitetura do *framework*, é utilizado *template metaprogramming* de C++

para extrair propriedades dos tipos de dado do usuário e integrá-los com as estruturas de dados geométricas definidas. Evitando o uso de programação orientada a objetos, o *framework* tenta não possuir *overhead* em tempo de execução e não ser intrusivo aos dados do usuário, não necessitando de herança a tipos internos à biblioteca. Desta forma, observa-se duas áreas distintas: o módulo de personalização de dados, onde são definidas e detectadas propriedades dos dados do usuário e o módulo de estruturas geométricas em si. Além destes dois módulos, dois tipos de interface são propostos: uma interface procedural e uma interface funcional.

Outros submódulos são implementados para uso interno do *framework*. Pode-se citar o módulo de algoritmos e o módulo de memória, que contém funções cujo propósito é promover otimizações específicas para algumas estruturas de dados. Porém, estes algoritmos e funções de memória são expostos para o uso externo, caso o usuário ache necessário.

O *framework* é implementado em C++17. Isso possibilita o uso de algumas técnicas de programação novas na linguagem, tais como fold expression, variadic templates, aligned_storage, constexpr, lambda e auto. Essas novas construções de linguagem facilitam no desenvolvimento de bibliotecas e tornam possíveis o uso de algumas técnicas novas de implementação.

4.1.1 Módulo de personalização de dados

Para o módulo de personalização de dados, é definido o namespace `geometricks::dimensions`. Neste módulo, o usuário provê instanciações de *traits* para o seu tipo de dado, expondo dimensões e características quando necessário. Este módulo também implementa protocolos para detectar e escolher métodos de acesso às diferentes dimensões dos tipos de dado. Como exemplo, um usuário pode definir métodos `get(const T&, geometricks::dimensions::dimension_t<I>)` para acessar a dimensão I do seu dado de tipo `T` dentro do próprio namespace de definição do dado. Caso não seja encontrado este método, a biblioteca procura por um método `get<I>(const T&)`. Outra possível forma de extração de dimensão que tem precedência às outras duas é a definição de um método estático `_` numa especialização da `struct get` definida dentro do namespace `geometricks::dimensions::get_customization`. Esta convenção de personalização de estruturas e algoritmos é utilizada no resto do *framework* e são exploradas mais a fundo na seção 4.2.2.

Este módulo ainda possui utilitários para os módulos de estruturas e algoritmos e para extensão dos dados em forma de `structs` de formas geométricas específicas e tipos de dados que definem dimensões em tempo de compilação, de forma que a primeira dimensão tenha um tipo diferente de dado que a segunda. Cada dimensão D desta forma é definida como uma instância do tipo `dimension_t<D>`, e pode ter seu valor acessado por `dimension_v<D>`. As `structs` definidas para formas geométricas incluem `rectangle`,

`point`, `segment`, `polygon` e `volume`. Como as estruturas do *framework* trabalham apenas com essas formas, o usuário deve prover *traits* capazes de converter seus dados para estes tipos específicos, fazendo uma ponte para o uso da biblioteca.

Os *traits* possíveis de personalização implementados permitem expor o número de dimensões, no caso dos *traits* definidos por especializações da `struct geometricks::dimensions::dimensional_traits` ou informações de forma geométrica do dado e como extrair objetos definidos pelo *framework*, tais como polígonos, retângulos e retas, a partir dos tipos de dado definidos pelo usuário. Os *traits* de extração de forma geométrica podem, desta forma, ser passados como parâmetros para as estruturas, permitindo que um tipo de dado se adapte as diferentes formas geométricas caso necessário. Algumas especializações padrão já vem conectadas diretamente ao *framework*, tais como adaptadores para dados presentes na std lib do C++ 17 e *arrays* nativos da linguagem.

A técnica de implementação *traits* é explorada na seção 4.3.3 deste capítulo.

4.1.2 Módulo de estrutura de dados

O módulo de estrutura de dados pode ser dividido em duas partes: as estruturas de dados usadas internamente para otimização de algoritmos e armazenamento e as estruturas de dados expostas externamente ao usuário final. Dados do usuário são personalizados de acordo com as definições propostas na seção 4.1.1 e então podem ser armazenados nas diversas estruturas de dados presentes. Cada estrutura de dados no *framework* visa uma implementação genérica e de fácil usabilidade, expondo apenas métodos necessários aos usuários e resolvendo a maior quantidade possível de operações em tempo de compilação, utilizando de *template metaprogramming*. Isso causa um *trade-off* de tempo maior de compilação, mas tem-se um ganho de desempenho e espaço de armazenamento.

4.1.3 Módulo de algoritmos

O módulo de algoritmos do *framework* introduz métodos para tratar tipos de dados genéricos passados pelo usuário às estruturas de dados, assim como funções com especificações constexpr usadas nos *templates*. Além disso, são disponibilizadas métricas de distância euclidiana e de Manhattan para serem utilizadas com os tipos de dados do usuário sem a necessidade de personalização. O módulo visa de maneira geral disponibilizar os algoritmos como ponto de personalização para o usuário, de forma que implementações mais otimizadas e definidas pelo usuário sejam preferidas às implementações definidas pela biblioteca.

Entre os métodos disponibilizados, tem-se pontos de personalização para operações de calcular o valor absoluto de um número, com um método `abs`; a diferença absoluta entre dois números, com um método `absolute_difference`; uma função iter_swap para

substituir o valor contido em 2 iteradores e um *trait* `zero_traits` utilizado em funções internas da biblioteca para calcular a média.

A seção 4.2.4 contém exemplos de código para integrar tipos de dados do usuário com os algoritmos definidos pela biblioteca.

4.1.4 Módulo de memória

O módulo de memória do *framework* separa as estruturas de dados do método de alocação de memória das estruturas. Para tanto, são definidas classes dinamicamente polimórficas não intrusivas. Outros *frameworks*, tais como a std lib do C++, definem o alocador de memória como uma *policy trait* em tempo de compilação, ganhando desempenho extra em troca de cada combinação de estrutura/alocador de memória ter um tipo de dado diferente, dificultando a usabilidade. Dito isto, o *framework* opta por melhor usabilidade ao invés de desempenho máximo.

Para este módulo, é definida uma interface utilizando-se de *type erasure* para criar *views* a alocadores de memória genéricos. Esse tipo de dado pode receber qualquer alocador que disponha de métodos de alocação e desalocação de memória. Para a alocação, pode-se definir métodos de alocação com alinhamento ou sem alinhamento de memória e, para ambos os casos, o *framework* utiliza um protocolo para procurar pelo método certo. Primeiro é procurado por um método `allocate` estático numa especialização da `struct allocator` no namespace `geometricks::memory::allocator_customization`. Caso esse método não seja encontrado, são procurados métodos `allocate` ou `alloc` no dado. Por fim, se nenhum destes métodos for encontrado, são procuradas funções `allocate` ou `alloc` que recebem o alocador de memória. Se o alocador de memória permitir alocação com alinhamento, este método é preferido sobre o método sem alinhamento.

O desalocador de memória trabalha com um conceito parecido. É possível definir funções para desalocar memória com ou sem o tamanho do dado. Primeiro é procurado um método `deallocate` estático numa especialização da `struct allocator` no namespace `geometricks::memory::allocator_customization`. Caso esse método não for encontrado, são procuradas funções `deallocate` ou `dealloc` que recebem o alocador de memória. Por fim, se nenhum destes métodos seja encontrado, são procuradas funções `deallocate` ou `dealloc` que recebem o alocador.

A seção 4.2.5 contém exemplos de criação de alocadores de memória personalizados e como diferentes alocadores podem ser utilizados pelo *framework*. Também são disponibilizadas funções para definir o alocador de memória padrão e usar o alocador padrão.

4.2 Usabilidade

Esta seção trata da usabilidade dos módulos do *framework*, contendo trechos de código de como um usuário pode integrar seu tipo de dado com as estruturas definidas pelo *framework*. As seções a seguir usam como exemplo uma classe de pontos 2D definida no exemplo 4.1, integrando instâncias da classe com as diferentes estruturas da biblioteca.

Exemplo 4.1 – Definição de estrutura de ponto 2D

```
1 struct point{
2     int x, y;
3 };
```

4.2.1 Usando as estruturas

Uma estrutura de dados geométrica pode ser inicializada a partir de seu construtor. Todas as estruturas do *framework* possuem um construtor que recebe dois iteradores para um conjunto de entrada de dados: o primeiro apontando para o primeiro elemento e o segundo apontando para um depois do último elemento. Este construtor realiza o processo de *bulk loading* no caso da estrutura permitir otimizações deste tipo ou apenas inserções sequenciais no caso da estrutura não permitir. Um exemplo de uso de tal construtor é observado no exemplo 4.2, onde uma árvore KD é construída a partir de um vetor de pontos de entrada definidos de acordo com a estrutura no exemplo 4.1. As linhas 3 a 8 populam um `std::vector<point>` com instâncias de `point` calculadas de forma externa na linha 6. A árvore então é construída na linha 9 a partir dos dados de entrada, utilizando um par de iteradores. Estruturas que suportam inserções, tais como *quadtrees*, *octrees* e *rtrees* também definem um construtor padrão que apenas inicializa o alocador e a estrutura básica da estrutura. Caso seja necessário outros parâmetros para a inicialização, tais como o retângulo que engloba a árvore inteira numa raiz de *quadtree*, outros construtores são definidos recebendo esses parâmetros com valores não padrão. Por fim, em cada construtor, é possível passar como argumento *views* para alocadores de memória personalizados. Estes construtores são exemplificados no exemplo 4.3. As linhas 3 e 4 constroem uma *quadtree* contida no espaço $[(-10,-10),(10,10)]$. A linha 5 constrói uma *quadtree* no espaço $[(-max\_int,-max\_int),(max\_int,max\_int)]$. As linhas 6 a 8 constroem uma árvore por *bulk loading*, com a linha 6 calculando os pontos de entrada externamente e as linhas 7 e 8 construindo a árvore. A linha 9 define um alocador personalizado que aloca objetos na *stack*, não definido neste exemplo. As linhas 10 a 12 criam uma *quadtree* utilizando este alocador.

O exemplo 4.2 também exemplifica como operações de vizinho mais próximo, k vizinhos mais próximos e busca por segmento podem ser realizadas em uma árvore KD. A linha 10 calcula um `point` de forma externa. A linha 11 realiza uma busca de vizinho mais

próximo usando este ponto, retornando o ponto mais próximo e a distância calculada. A linha 12 define um k = 10, que é utilizado para uma busca de k vizinhos mais próximos na linha 13 e 14, retornando um vetor contendo 10 pares de pontos e distância, ordenados de menor para maior distância. As linhas 15 a 18 demonstram como uma busca por segmento pode ser realizada. A linha 15 define o ponto inferior do segmento, enquanto a linha 16 define o ponto superior do segmento. As linhas 17 e 18 realizam uma busca por segmento e retornam um vetor contendo todos os pontos contidos por este dentro da árvore.

A seção 4.2.2 demonstra os passos necessários para integrar a `struct point` com o *framework*.

Exemplo 4.2 – Construção de Árvore KD e busca nearest neighbor

```
1  #include <geometricks/data_structure/kd_tree.hpp>
2  int use_kd_tree() {
3    int samples = 100;
4    std::vector<point> input{};
5    for(int i = 0; i < samples; ++i) {
6      point p = ...;
7      input.push_back( p );
8    }
9    geometricks::kd_tree<point> tree{ input.begin(), input.end() };
10   point query_point = ...;
11   auto [nearest,distance] = tree.nearest_neighbor( query_point );
12   int k = 10;
13   std::vector<std::pair<point,uint64_t>> output_vector =
14                       tree.k_nearest_neighbor( query_point, k );
15   point min_point = { -100, -100 };
16   point max_point = { +100, +100 };
17   std::vector<point> range =
18                       tree.range_search( min_point, max_point );
19   return 0;
20 }
```

Exemplo 4.3 – Construção de estrutura *quadtree*

```
1  #include <geometricks/data_structure/quad_tree.hpp>
2  int construct_quad_tree() {
3    geometricks::quad_tree<point> rect_construct_tree{
4                        geometricks::rectangle{ -10, -10, 10, 10 } };
5    geometricks::quad_tree<point> default_tree;
6    std::vector<point> input_vector = get_samples();
7    geometricks::quad_tree<point> bulk_loaded_tree{
8                        input_vector.begin(), input_vector.end() };
9    stack_allocator<1000> alloc;
10   geometricks::quad_tree<point> custom_alloc{ alloc };
11   return 0;
12 }
```

4.2.2 Personalização de dados usuário

Esta seção trata da definição de *traits* e especializações para integrar a `struct point` definida no exemplo 4.1 com as estruturas do *framework*.

4.2.2.1 Definindo dimensões do dado

Para integrar um dado com as diferentes estruturas de dados presentes no *framework*, primeiro deve-se definir o número de dimensões do dado. Para a estrutura de `point` definida no exemplo 4.1, são definidas duas dimensões para o dado: a dimensão `x` e a dimensão `y`. O número de dimensões contidos nesta estrutura é exposto ao *framework* conforme o exemplo 4.4. As linhas 3 a 5 definem uma especialização de *template* da `struct dimensional_traits` para a `struct point`. Dentro dessa especialização, na linha 4 é definido um valor de uma constante estática constexpr, utilizada em tempo de compilação na estrutura de árvore KD como o número de dimensões utilizados por árvores para este tipo de dado.

Exemplo 4.4 – Declaração de dimensões da estrutura point

```
1  #include <geometricks/data_structure/dimensional_traits.hpp>
2  namespace geometricks::dimension {
3    struct dimensional_traits<point> {
4      static constexpr int dimensions = 2;
5    };
6  }
```

4.2.2.2 Expondo as dimensões do dado

Para expor as dimensões dos dados aos algoritmos e às estruturas de dados, o *framework* segue um protocolo de diferentes preferências a funções definidas pelo usuário conforme o exemplo 4.5. A dimensão 0 é definida como o valor `x` e a dimensão 1 é definida como o valor `y` de cada ponto. A preferência de chamada segue a seguinte ordem, por dimensão:

1. Procura-se por um método estático `_` retornando o valor contido naquela dimensão em uma especialização da `struct get` definida no namespace `geometricks::dimension::get_customization` que deve receber uma instância do dado e uma instância de um *tag type* do tipo `geometricks::dimension::dimension_t<N>`, onde N é a dimensão atual. Isso pode ser visto tanto para a dimensão 0 quanto para a dimensão 1 nas linhas 1 a 12, retornando `x` e `y` respectivamente.

2. Caso uma especialização definida acima não seja encontrada, testa-se por uma função utilizando ADL livre que recebe como parâmetros uma instância do dado e uma instância de um *tag type* do tipo `geometricks::dimension::dimension_t<N>`, onde N é a dimensão do dado. Esse método pode ser observado nas linhas 13 a 20.

3. Se não for possível realizar uma chamada com nenhuma das funções acima, testa-se por uma função livre que recebe um inteiro N, onde N é a dimensão atual, como parâmetro de *template* e uma instância do dado. Esse método é observado nas linhas 21 a 30.

4. Por fim, caso não seja possível realizar a chamada com nenhum destes métodos, ocorre um erro de compilação.

Exemplo 4.5 – Definição de métodos de acesso às dimensões de estrutura *point*

```
1  namespace geometricks::dimension::get_customization {
2    struct get<point> {
3      static int _( const point& p,
4                    geometricks::dimension::dimension_t<0> ) {
5        return p.x;
6      }
7      static int _( const point& p,
8                    geometricks::dimension::dimension_t<1> ) {
9        return p.y;
10     }
11   };
12 }
13 int get( const point& p,
14           geometricks::dimension::dimension_t<0> ) {
15   return p.x;
16 }
17 int get( const point& p,
18           geometricks::dimension::dimension_t<1> ) {
19   return p.y;
20 }
21 template<int I>
22 int get( const point& p );
23 template<>
24 int get<0>( const point& p ) {
25   return p.x;
26 }
27 template<>
28 int get<1>( const point& p ) {
29   return p.y;
30 }
```

4.2.2.3 Funções de distância personalizadas

Uma função de distância personalizada pode ser passada às funções de vizinho mais próximo. Para definir uma função personalizada para a busca, uma `struct` que contém um operador () para comparar dois pontos entre si e entre diferentes dimensões é utilizada. Isso pode ser visto conforme o exemplo 4.6, que define uma função de distância euclidiana para dados do tipo `point` definidos no exemplo 4.1. As linhas 3 a 7 são obrigatórias,

definindo a distância entre dois pontos. As linhas 8 a 36 definem diferentes métodos para calcular a distância de um ponto ao hiperplano, conforme o seguinte protocolo:

1. Testa-se por um operador() que recebe duas instâncias do tipo de dado e uma instância de um *tag type* do tipo `geometricks::dimension::dimension_t<N>`, onde N é a dimensão atual. Isso é observado para a dimensão 0 nas linhas 8 a 13 do exemplo 4.6.

2. Caso esse operador não seja encontrado, testa-se por um operador() que recebe um tipo de dado, um tipo de valor presente naquela dimensão e uma *tag type* do tipo `geometricks::dimension::dimension_t<N>`. No exemplo 4.6, este valor é do tipo inteiro, devido ao tipo de `x` ser um `int` e N = 0. Isso pode ser observado nas linhas 14 a 19. O valor passado como segundo argumento a essa função é o resultado de `geometricks::dimension::get(p2, dimension_v<N>)`, definido na seção 4.2.2.2.

3. Caso nenhuma das alternativas acima seja encontrada, é testado mais uma vez a alternativa acima, apenas mudando a ordem entre ponto e valor nos parâmetros. Isso pode ser observado nas linhas 20 a 25.

4. Se esse método não for possível, testa-se por um operador() recebendo dois valores encontrados na dimensão N e um *tag type* do tipo `geometricks::dimension::dimension_t<N>`. Como argumento para essa função é passado o resultado de `geometricks::dimension::get(p1, dimension_v<N>)` e `geometricks::dimension::get(p2, dimension_v<N>)`. Isso pode ser observado nas linhas 26 a 31.

5. Se nenhum dos métodos acima for possível, define-se um método conforme as linhas 32 a 35 que recebe dois valores contidos em uma certa dimensão não especificada. Para cada dimensão N que essa for a única alternativa possível, chama-se esse método com os argumentos `geometricks::dimension::get(p1, dimension_v<N>)` e `geometricks::dimension::get(p2, dimension_v<N>)`.

6. Se nenhuma das alternativas acima for possível, ocorre um erro de compilação.

A definição do protocolo acima permite uma maior flexibilidade na definição das funções de distância, sendo possível definir diferentes métodos para comparar dimensões diferentes ou apenas manter uma simples comparação padrão.

Exemplo 4.6 – Definição de função de distância personalizada

```
1  #include <geometricks/data_structure/dimensional_traits.hpp>
2  struct point_distance {
3    int operator()( const point& p1, const point& p2 ) {
4      int x_diff = std::abs(p1.x - p2.x);
5      int y_diff = std::abs(p1.y - p2.y);
6      return x_diff * x_diff + y_diff * y_diff;
7    }
8    int operator()(const point& p1,
9                   const point& p2,
10                  dimention_t<0> ) {
11     int x_diff = std::abs( p1.x - p2.x );
12     return x_diff * x_diff;
13   }
14   int operator()( const point& p1,
15                   int x,
16                   dimention_t<0> ) {
17     int x_diff = std::abs(p1.x - x);
18     return x_diff * x_diff;
19   }
20   int operator()( int x,
21                   const point& p2,
22                   dimention_t<0> ) {
23     int x_diff = std::abs(p2.x - x);
24     return x_diff * x_diff;
25   }
26   int operator()( int x1,
27                   int x2,
28                   dimention_t<0> ) {
29     int x_diff = std::abs(x1 - x2);
30     return x_diff * x_diff;
31   }
32   int operator()( int x1, int x2 ) {
33     int x_diff = std::abs(x1 - x2);
34     return x_diff * x_diff;
35   }
36 };
```

4.2.2.4 Criação de retângulos

Retângulos são implementados como uma `struct` que contém 4 valores inteiros de 64 bits: um valor `x` inicial; um valor `y` inicial; uma altura e uma largura. Objetos de ponto flutuante ou outros tipos de dado definidos pelo usuário devem realizar uma transformada de *viewport* para valores discretos ao criar seus retângulos.

Para criar um retângulo a partir de um objeto, usa-se o ponto de personalização `geometricks::make_rectangle`. Esse método utiliza o seguinte protocolo para procurar por uma função do usuário, conforme o exemplo 4.7:

1. Um método estático `_` que recebe um `const T&` e retorna um `geometricks::rectangle` presente em uma especialização da `struct make_rectangle` no namespace `geometricks::rectangle_customization`. Isso pode ser visto nas linhas 17 a 23.

2. Uma função `make_rectangle` que recebe um `const T&` e um *tag type* do tipo `geometricks::rectangle_tag` e retorna um `geometricks::rectangle`. Esse método pode ser observado nas linhas 24 a 27.

3. Um cast do retorno de uma chamada de um método `bounding_rect()` presente no tipo de dado para um `geometricks::rectangle`. Esse método é definido nas linhas 7 a 9.

4. Um cast de uma instância do tipo de dado para um `geometricks::rectangle`. Essa função está presente nas linhas 10 a 12.

5. Um `geometricks::rectangle` construído pelos métodos `get<I>`, com I variando de 0 a 3. Essa função é definida nas linhas 28 a 31.

6. Um `geometricks::rectangle` construído pelos resultados das chamadas de operadores [I], com I variando de 0 a 3. Esse método está definido nas linhas 13 a 16.

7. Caso nenhum dos métodos acima seja possível, um erro de compilação é gerado.

Exemplo 4.7 – Exemplo de criação de retângulo

```
1  #include <geometricks/data_strucutre/shapes.hpp>
2  struct rect {
3    int64_t x_left;
4    int64_t y_top;
5    int64_t width;
6    int64_t height;
7    geometricks::rectangle bounding_rect() const {
8      return geometricks::rectangle{ x_left, y_top, width, height };
9    }
10   operator geometricks::rectangle() const {
11     return this->bounding_rect();
12   }
13   int64_t operator[]( size_t pos ) const {
14     return reinterpret_cast<const int64_t*>( this )[ pos ];
15   }
16 };
17 namespace geometricks::rectangle_customization {
18   template<> struct make_rectangle<rect> {
19     static geometricks::rectangle _( const rect& value ) {
20       return value.bounding_rect();
21     }
22   }
23 };
24 geometricks::rectangle make_rectangle( const rect& rect,
25                                        geometricks::rectangle_tag ) {
26   return rect.bounding_rect();
27 }
28 template<int I>
29 int64_t get( const rect& value ) {
30   return value[ I ];
31 }
```

Estes retângulos então são utilizados para armazenar os objetos dentro de uma *quadtree* ou *rtree*.

4.2.2.5 Criação de volumes

Volumes, assim como retângulos, são implementados como uma `struct` utilizando inteiros de 64 bits. Porém, diferente de um retângulo, volumes possuem 6 valores: um valor `x` inicial; um valor `y` inicial; um valor `z` inicial, uma altura, uma largura e uma profundidade. Objetos de ponto flutuante ou outros tipos de dados definidos pelo usuário devem realizar

uma transformada de *viewport* para valores discretos ao criar seus retângulos.

Para criar um volume a partir de um objeto, usa-se o ponto de personalização `geometricks::make_volume`. Esse método utiliza o seguinte protocolo para procurar por uma função do usuário, conforme o exemplo 4.8:

1. Um método estático `_` que recebe um `const T&` e retorna um `geometricks::volume` presente em uma especialização da `struct make_volume` no namespace `geometricks::volume_customization`. Observado nas linhas 20 a 26.

2. Uma função `make_volume` que recebe um `const T&` e um *tag type* do tipo `geometricks::volume_tag` e retorna um `geometricks::volume`. Esse método é visto nas linhas 27 a 30.

3. Um cast do retorno de uma chamada de um método `bounding_box()` presente no tipo de dado para um `geometricks::volume`. Definido nas linhas 9 a 12.

4. Um cast de uma instância do tipo de dado para um `geometricks::volume`. Presente nas linhas 13 a 15.

5. Um `geometricks::volume` construído pelos métodos `get<I>`, com I variando de 0 a 5. Essa função pode ser observada nas linhas 31 a 34.

6. Um `geometricks::volume` construído pelos resultados das chamadas de operadores [I], com I variando de 0 a 5. Esse método é definido nas linhas 16 a 19.

7. Caso nenhum dos métodos acima seja possível, um erro de compilação é gerado.

Exemplo 4.8 – Exemplo de criação de volume

```
#include <geometricks/data_strucutre/shapes.hpp>
struct box {
  int64_t x_left;
  int64_t y_top;
  int64_t z_back;
  int64_t width;
  int64_t height;
  int64_t depth;
  geometricks::volume bounding_box() const {
    return geometricks::volume{ x_left, y_top, z_back,
                                width, height, depth };
  }
  operator geometricks::volume() const {
    return this->bounding_box();
  }
  int64_t operator[]( size_t pos ) const {
    return reinterpret_cast<const int64_t*>( this )[ pos ];
  }
};
namespace geometricks::volume_customization {
  template<> struct make_volume<rect> {
    static geometricks::volume _( const box& value ) {
      return value.bounding_box();
    }
  }
};
geometricks::volume make_volume( const box& box,
                                 geometricks::volume_tag ) {
  return box.bounding_box();
}
template<int I>
int64_t get( const box& value ) {
  return value[ I ];
}
```

Estes volumes então são utilizados para armazenar os objetos dentro de uma *octree*.

4.2.3 Personalização de estruturas

Algumas estruturas podem ter parâmetros de *template* diferentes para mudar um pouco seu comportamento utilizando de *policy traits*. Uma árvore KD, por exemplo, pode receber diferentes comparadores como parâmetro de *template* para separar os dados pelo hiperplano. Como exemplo, pode-se ver um comparador personalizado no exemplo 4.9. As linhas 1 a 5 definem um comparador personalizado, que jogam os maiores elementos para a direita e os menores elementos para a esquerda, organizando a árvore no oposto da sua organização usual. A linha 7 calcula um vetor de forma externa e as linhas 9 e 10 constroem uma árvore utilizando este vetor. Uma *quadtree* pode ser personalizada conforme a sua profundidade máxima e número de elementos por nó. Caso uma *quadtree* chegue a profundidade máxima, ela para de subdividir o nó e apenas mantém uma lista encadeada contendo até infinitos elementos. Uma personalização pode ser observada no exemplo 4.10. As linhas 2 a 5 definem uma `struct` contendo *traits* para uma profundidade máxima de 100 e uma quantidade máxima de elementos por folha de 8. A profundidade máxima é definida em uma constante estática constexpr do tipo `int max_depth`. A quantidade de elementos por folha é definido em uma constante estática constexpr do tipo int leaf_size. As linhas 6 a 9 utilizam o construtor padrão da árvore personalizada com esse *trait*.

Exemplo 4.9 – Definição de função para árvore KD de split de ponto em dimensões específicas

```
1  struct point_compare {
2    bool operator()(int cur_dimension1, int cur_dimension2) {
3      return cur_dimension1 > cur_dimension2;
4    }
5  };
6  int construct_customized_kd_tree() {
7    std::vector<point> input = ...;
8    ...
9    geometricks::kd_tree<point, point_compare>
10         tree{input.begin(), input.end()};
11   ...
12   return 0;
13 }
```

Exemplo 4.10 – *Quadtree* utilizando parâmetros personalizados

```
1 #include <geometricks/data_structure/quad_tree.hpp>
2 struct my_quad_tree_traits {
3   static constexpr int max_depth = 100;
4   static constexpr int leaf_size = 8;
5 };
6 int construct_quad_tree() {
7   geometricks::quad_tree<point, my_quad_tree_traits> tree{};
8   return 0;
9 }
```

4.2.4 Personalização de algoritmos auxiliares

Para exemplificar algoritmos nessa seção, será assumida uma implementação de uma classe `bit_num` com a interface definida no exemplo 4.11. O exemplo omite outras operações matemáticas e operações binárias por brevidade, mas assume que a `struct` implemente todas as operações implementadas por um dado do tipo `int`.

Exemplo 4.11 – Interface de uma classe big_num

```
1  struct big_num {
2    big_num(int64_t val);
3    big_num(std::string_view str);
4    big_num(const big_num& rhs);
5    ...
6    big_num& operator-=(const big_num& rhs);
7    big_num& operator-=(int64_t rhs);
8    big_num& operator+=(const big_num& rhs);
9    big_num& operator+=(int64_t rhs);
10   ...
11 };
```

O exemplo 4.12 demonstra como essa `struct` pode ser integrada com os algoritmos do *framework*. As linhas 3 a 5 definem uma função num ponto de personalização para calcular o valor absoluto de um tipo de dado genérico. O protocolo para este algoritmo funciona da seguinte forma:

1. Método estático _ que recebe um valor e retorna o valor absoluto.

2. Conversão do dado para `size_t`.

3. Verifica se o valor é menor que o `int` 0. Caso ele seja, retorna o `operator-` do dado. Caso contrário, retorna o valor.

As linhas 10 a 12 definem um método para calcular a diferença absoluta entre dois `big_num`. Este algoritmo checa por uma especialização da `struct absolute_difference` no namespace `geometricks::algorithm::absolute_difference_customization`. Caso uma especialização não seja encontrada, é chamado o método `abs` no resultado do `operator-` binário entre os dois valores.

Exemplo 4.12 – Personalização de algoritmos para big_num

```
1  namespace geometricks::algorithm::abs_customization {
2    template<> struct abs<big_num> {
3      static big_num _( const big_num& value ) {
4        return value < 0 ? -value : value;
5      }
6    }
7  }
8  namespace geometricks::algorithm::absolute_difference_customization {
9    template<> struct absolute_difference<big_num> {
10     static big_num _( const big_num& lhs, const big_num& rhs ) {
11       return geometricks::algorithm::abs( lhs - rhs );
12     }
13   }
14 }
```

Outro algoritmo implementado como ponto de personalização é iter_swap. Apesar da std lib permitir a personalização do algoritmo swap, não é possível personalizar o algoritmo iter_swap. Devido ao *framework* necessitar a personalização de alguns iteradores, esse algoritmo é implementado como ponto de personalização. Isso pode ser exemplificado conforme o exemplo 4.13. As linhas 1 a 8 definem uma `struct` de um iterador para um inteiro. As linhas 11 a 14 definem uma função que troca o valor contido entre dois valores destes iteradores. Para a realização da operação, o *framework* segue o seguinte protocolo:

1. Procura por uma especialização da `struct iter_swap` para o tipo de iterador passado como primeiro argumento que seja capaz de fazer a troca entre os dois iteradores.

2. Procura por uma especialização da `struct iter_swap` para o tipo de iterador passado como segundo argumento que seja capaz de fazer a troca entre os dois iteradores.

3. Chama `iter_swap` por ADL, tendo *fallback* no algoritmo `std::iter_swap` caso não seja encontrado.

4. Chama `swap` por ADL nos valores dos iteradores, tendo *fallback* no algoritmo `std::swap` caso não seja encontrado.

Exemplo 4.13 – Personalização de iterador

```
1  struct int_iterator {
2    int* m_iter;
3    int_iterator& operator++();
4    int& operator*();
5    int* operator->();
6    int_iterator& operator+=(size_t);
7    ...
8  };
9  namespace geometricks::algorithm::iter_swap_customization {
10   template<> struct iter_swap<int_iterator> {
11     static void _( int_iterator& lhs, int_iterator& rhs ) {
12       using std::swap;
13       return swap( *lhs.m_iter, *rhs.m_iter );
14     }
15   };
16 }
```

O *framework* ainda provê alguns outros algoritmos de uso interno, mas que não são pontos de personalização para o usuário final e portanto não serão mencionados nesta seção.

4.2.5 Alocadores de memória personalizados

Esta seção demonstra como pode ser implementado um simples alocador de memória alinhada usando a chamada aligned_alloc da biblioteca C, como esse alocador de memória é utilizado com as estruturas de dados e como mudar o alocador de memória padrão das estruturas para o alocador definido nos exemplos.

Primeiro, define-se uma estrutura capaz de realizar alocações alinhadas conforme o exemplo 4.14. A alocação de memória em si é definida nas linhas 4 a 6 e a desalocação de memória nas linhas 10 a 12. O resto do exemplo trata de outros métodos para definir as funções. As funções de alocação de memória têm preferência na seguinte ordem:

1. Método estático `allocate` definido numa especialização de *template* da `struct allocator` no namespace `geometricks::memory::allocator_customization`. Definida nas linhas 19 a 22.

2. Método `allocate` definido dentro da classe. Definido nas linhas 4 a 6.

3. Método `alloc` definido dentro da classe. Definido nas linhas 7 a 9.
4. Função `allocate` que recebe um objeto da classe como primeiro argumento. Definido nas linhas 28 a 30.
5. Função `alloc` que recebe um objeto da classe como primeiro argumento. Definido nas linhas 32 a 34.

Caso não seja possível alocar com um dos métodos de maior prioridade, tenta-se outros métodos com prioridade menor. Por fim, caso não seja possível nenhum dos métodos, ocorre um erro de compilação.

Para a desalocação de memória, a preferência segue a seguinte ordem:

1. Método estático `deallocate` definido numa especialização de *template* da `struct allocator` no namespace `geometricks::memory::allocator_customization`. Definida nas linhas 23 a 25.
2. Método `deallocate` definido dentro da classe. Definido nas linhas 10 a 12.
3. Método `dealloc` definido dentro da classe. Definido nas linhas 13 a 15.
4. Função `deallocate` que recebe um objeto da classe como primeiro argumento. Definido nas linhas 35 a 37.
5. Função `dealloc` que recebe um objeto da classe como primeiro argumento. Definido nas linhas 38 a 40.

Assim como com os métodos de alocação de memória, a ordem segue a maior preferência até a menor preferência. Caso não seja encontrado nenhum método de desalocação, ocorre um erro de compilação.

Exemplo 4.14 – Estrutura com método definido para alocação e desalocação de memória alinhada

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  struct aligned_alloc_allocator {
4    void* allocate( size_t sz, size_t align ) {
5      return aligned_alloc( align, sz );
6    }
7    void* alloc( size_t sz, size_t align ) {
8      return allocate( sz, align );
9    }
10   void deallocate( void* ptr ) {
11     free( ptr );
12   }
13   void dealloc( void* ptr ) {
14     deallocate( ptr );
15   }
16 };
17 namespace geometricks::memory::allocator_customization {
18   template<> struct allocator<aligned_alloc_allocator> {
19     static void* allocate( aligned_alloc_allocator& alloc, size_t sz,
20                            size_t align ) {
21       return alloc.allocate( sz, align );
22     }
23     static void deallocate( aligned_alloc_allocator& alloc, void* ptr ) {
24       alloc.deallocate( ptr );
25     }
26   };
27 }
28 void* allocate( aligned_alloc_allocator& alloc, size_t sz,
29                 size_t align ) {
30   return alloc.allocate( sz, align );
31 }
32 void* alloc( aligned_alloc_allocator& alloc, size_t sz, size_t align ) {
33   return alloc.allocate( sz, align );
34 }
35 void deallocate( aligned_alloc_allocator& alloc, void* ptr ) {
36   alloc.deallocate( ptr );
37 }
38 void dealloc( aligned_alloc_allocator& alloc, void* ptr ) {
39   alloc.deallocate( ptr );
40 }
```

Esse alocador de memória pode ser utilizado com qualquer estrutura passando uma referência como último argumento ao construtor conforme o exemplo 4.15. A linha 3 constrói o alocador de memória, a linha 4 define o retângulo da raiz da *quadtree* e a linha 5 constrói a *quadtree* passando o retângulo da raiz e o alocador de memória da árvore.

Exemplo 4.15 – Construindo QuadTree com malloc alinhado

```
1 int quad_tree_aligned_malloc_example() {
2   using qtree = geometricks::quad_tree<std::tuple<int,int,int,int>>;
3   aligned_alloc_allocator allocator{};
4   geometricks::rectangle tree_rect{ -1000, -1000, 2000, 2000 };
5   qtree tree{ tree_rect, allocator };
6   return 0;
7 }
```

Uma estrutura construída sem receber como argumento um alocador de memória usa o alocador de memória padrão. Para mudar o alocador padrão, observa-se o exemplo 4.16. As linhas 3 e 4 salvam o alocador padrão atual na variável `def_alloc`. A linha 6 muda o alocador padrão para o alocador `allocator` definido na linha 5. A linha 7 cria o retângulo da raiz da *quadtree*. A linha 8 cria a árvore usando o novo alocador padrão. A linha 9 restaura o alocador padrão para `def_alloc`.

Exemplo 4.16 – Mudança de alocador padrão

```
1 int change_default_allocator_example() {
2   using qtree = geometricks::quad_tree<std::tuple<int,int,int,int>>;
3   geometricks::allocator def_alloc =
4     geometricks::memory::get_default_allocator();
5   aligned_alloc_allocator allocator{};
6   geometricks::memory::set_default_allocator( allocator );
7   geometricks::rectangle tree_rect{ -1000, -1000, 2000, 2000 };
8   qtree tree{ tree_rect };
9   geometricks::memory::set_default_allocator( def_alloc );
10  return 0;
11 }
```

4.3 Implementação

Esta seção traz técnicas de implementação gerais e específicas a cada estrutura. São aqui apresentadas técnicas gerais utilizando ferramentas da linguagem C++ e técnicas específicas às diferentes estruturas, com uma introdução a sua necessidade, definições e exemplos.

4.3.1 SFINAE

Em C++, na definição de diferentes classes genéricas, pode-se ter a necessidade de incluir diferentes métodos dependendo das propriedades do objeto. Como motivação, são citados 2 exemplos.

1. Um container que não vê a necessidade de criar um destrutor mas deseja armazenar objetos de tipo genérico, tendo necessidade de chamar placement new e o destrutor manualmente, o que requer a definição de um destrutor e um operador =. Porém, caso o tipo de dado T seja trivially_copyable e trivially_destructible, esses métodos podem ser gerados automaticamente pelo compilador, abrindo possíveis otimizações tais como habilitando o *container* a ser utilizado com memcpy [Cpp Reference Trivially Copyable].

2. Em um ponto de personalização, um objeto genérico pode definir um dentre diversos métodos para realizar uma operação específica. Para resolver esse problema, precisa-se testar quais destes métodos são definidos para o objeto e apenas lançar um erro caso nenhuma das alternativas esteja presente. Caso mais de uma alternativa seja possível, um protocolo deve ser definido para selecionar uma entre elas. Isso deve ser resolvido em tempo de compilação.

Substitution Failure Is Not An Error (SFINAE) [Vandevoorde e Josuttis 2003] abusa o fato do compilador tentar sempre instanciar o *template* mais especializado e, caso não seja possível instanciar este *template*, esta alternativa é simplesmente descartada ao invés de resultar em um erro de compilação. Desta forma, esta técnica pode ser empregada para detectar a presença de um método dentro de uma classe e escolher diferentes implementações conforme o exemplo 4.17, onde é definido um protocolo para escolher um entre dois diferentes métodos de alocação de memória e causar um erro de compilação em caso contrário. Nas linhas 3 e 4, é definida uma `struct` que recebe dois parâmetros por *template*, um parâmetro de um tipo genérico `T` e outro parâmetro que não é utilizado e tem seu valor padrão igual a `void`. Esta `struct` herda de false_type, sendo esta instanciação que define a não presença de um método `allocate`. Depois, nas linhas 5 a 8, uma especialização dessa `struct` tenta chamar o método `allocate` em uma declaração de `T` e utiliza void_t para jogar o retorno da função para `void`. Esta segunda definição mais especializada herda de true_type. Segundo as regras de instanciação de *template*, uma instanciação dessa `struct` primeiro tenta as definições mais especializadas e, caso não seja possível instanciá-las, estas são apenas descartadas ao invés de gerar um erro de compilação. Desta forma, é possível condicionalmente herdar de true_type caso o método `allocate` seja presente e de false_type em caso contrário. Nas linhas 23 a 33, é definida uma `struct` usável como um objeto invocável que utiliza *branches* em tempo de compilação, instanciando o *template* definido anteriormente para verificar a existência ou

não do método. Caso o método não esteja presente, aquele trecho é descartado e testa-se por outras maneiras de realizar a operação.

Esta técnica é usada na biblioteca para definição de protocolos, extração de dimensões dos dados e otimizações específicas a tipos de dados diferentes.

Exemplo 4.17 – SFINAE usado para detecção de métodos de alocação de memória

```
1  namespace memory {
2    inline namespace mem_detail {
3      template<typename T, typename = void>
4      struct has_allocate_method : std::false_type {};
5      template<typename T>
6      struct has_allocate_method<T, std::voit_t<decltype(
7        std::declval<T>().allocate(std::declval<size_t>()))>>
8          : std::true_type{};
9      template<typename T>
10     constexpr bool has_allocate_method_v =
11       has_allocate_method<T>::value;
12     template<typename T, typename = void>
13     struct has_allocate_free_function : std::false_type {};
14
15     template<typename T>
16     struct has_allocate_free_function<T, std::voit_t<decltype(
17       allocate(std::declval<T>(), std::declval<size_t>()))>>
18         : std::true_type{};
19     template<typename T>
20     constexpr bool has_allocate_free_function_v =
21       has_allocate_free_function<T>::value;
22
23     constexpr inline struct allocate_fn {
24       template<typename T>
25       void* operator(const T& obj, size_t sz) {
26         if constexpr( has_allocate_method_v<T> ) {
27           return obj.allocate( sz );
28         }
29         else if constexpr( has_allocate_free_function_v<T> ) {
30           return allocate( obj, sz );
31         }
32       }
33     } allocate;
34   }
35 }
```

4.3.2 *Small Buffer Optimization*

Estruturas de dados nem sempre necessitam de um grande uso de memória. *Strings*, por exemplo, são usualmente pequenas. *Heaps* e vetores podem esperar, no caso médio, uma quantidade de memória específica, raramente necessitando mais que um número específico de elementos. Uma interface genérica de *type erasure*, técnica descrita na seção 4.3.4, recebe em média objetos de até certo tamanho. Por exemplo, um *type erasure* para funções deve receber no caso médio objetos do tamanho de um ponteiro, devido a este ser o espaço ocupado por um ponteiro para uma função.

Alocação dinâmica de memória requer um tempo extra não envolvido em alocação na pilha do programa. Chamadas a funções como malloc, mesmo que sejam otimizadas, envolvem estruturas de dados externas ao programa e chamadas ao sistema operacional ocasionalmente, assim como fragmentação de memória e não otimização da cache. Para evitar este impacto de desempenho, uma possível técnica é evitar alocações dinâmicas de memória mantendo um pequeno *buffer* na pilha do programa e, caso seja mais necessário mais memória, é feita uma chamada a funções de alocação de memória dinâmica. Na data atual, os compiladores da linguagem C++ GCC, Clang e Visual Studio implementam essa otimização para *strings* pequenas. Algumas bibliotecas, tais como o compilador [LLVM], possuem implementações de *small buffer optimization* (*SBO*) para vetores e mantém uma estrutura que não depende do parâmetro do tamanho do *buffer* num nível acima para facilitar a programação genérica, de forma que métodos externos recebam um ponteiro de uma superclasse de todos os diferentes *buffers* parâmetro, otimizando o tempo de compilação e tamanho do arquivo executável. Para este trabalho, esta abordagem foi escolhida para a implementação de estruturas de vetor, junto com o módulo de memória presente na sessão 4.1.4.

Small buffer optimization pode ser utilizado para buscas de *k nearest neighbor*, onde uma estrutura *heap* implementada em arranjo é utilizada para agilizar a busca. Guardando um pequeno *buffer* para um caso comum de k igual a 10, buscas por até 10 vizinhos são otimizadas sem a necessidade de alocação de memória dinâmica, melhorando o desempenho da função. Outro possível uso é na implementação de uma *stack* para simulação de chamadas recursivas, sendo possível otimizar a recursão em árvores num simples laço.

Implementações de *SBO* dependem da aplicação. Para esta seção, é demonstrado em código uma arquitetura geral de como funciona a técnica, conforme o exemplo 4.18, mesmo que a estrutura em si contenha variações em aplicações diferentes. Nas linhas 3 a 6 é definida a estrutura geral. Um dado pode se encontrar ou na *heap* ou na *stack*. Como o ponteiro para a *heap* e o espaço reservado para dados pequenos ocupam o mesmo espaço de memória, é definida uma union com os 2 membros. Na linha 7, é definida uma variável que indica qual membro da union está atualmente ativo. Dependendo da aplicação, esta

variável não é necessária, sendo o problema resolvível por dados externos à estrutura do *SBO*. Na linha 11, é definida uma função que recebe um tipo genérico de dado T e constrói este tipo na arquitetura. Na linha 13, testa-se se o tamanho do tipo de dado genérico recebido como argumento é maior que o tamanho do *buffer* em tempo de compilação. Caso o dado não caiba no *buffer*, as linhas 14 a 16 alocam memória na *heap*, constroem o dado, armazenam um ponteiro dentro do *buffer* e setam o campo ativo da union para `heap_storage`. As linhas 19 e 20 são compiladas caso o dado caiba no *buffer*, apenas construindo o objeto diretamente no espaço reservado e setando o membro ativo da union para `stack_storage`. Este exemplo é simplificado e não contém os dados necessários para extrair o tipo de dado armazenado no *buffer*, nem para desalocar a memória da *heap*.

Exemplo 4.18 – Arquitetura geral de estrutura small buffer optimization

```
1  template<size_t BufferSize, size_t Align = alignof(max_align_t)>
2  struct small_buffer_optimization {
3    union {
4      void* heap_storage;
5      alignas(Align) char[BufferSize] stack_storage;
6    };
7    bool is_stack;
8  };
9
10 template<typename T, size_t BufferSize>
11 void put(const T& value,
12           small_buffer_optimization<BufferSize>& storage) {
13   if constexpr(sizeof(T) > BufferSize) {
14     T* heap_value = new T(value);
15     storage.heap_storage = heap_value;
16     storage.is_stack = false;
17   }
18   else {
19     new (&storage.stack_storage) T(value);
20     storage.is_stack = true;
21   }
22 }
```

4.3.3 *Traits*

Em funções que recebem tipos de dados genéricos, é necessário extrair características destes dados para realizar operações diferentes. Por exemplo, para encontrar o valor mínimo presente numa coleção de objetos do tipo `T`, inicializamos primeiro uma variável contendo

o valor máximo que `T` pode suportar e atualizamos ela passo a passo com o algoritmo. Este valor máximo é definido em um *trait*, de forma que o código genérico inicializa a variável corretamente. Outro exemplo é com iteradores de diferentes estruturas de dados. Caso o iterador seja do tipo acesso aleatório, a distância entre dois iteradores I1 e I2 pode ser calculada com uma simples subtração, enquanto iteradores de memória não contígua precisam ser iterados para descobrir a distância entre os diferentes iteradores. Um *trait* para cada iterador é definido contendo o tipo de acesso do iterador de forma a otimizar chamadas quando possível.

Traits é uma técnica de implementação envolvendo o mecanismo de *template* do C++ para disponibilizar características e operações específicas a diferentes tipos de dados em tempo de compilação, criando meta funções usadas em código de programação genérica.

Uma meta função de *traits* para extrair informações de tipos numéricos e usada para calcular o valor mínimo em uma coleção pode ser definida conforme o exemplo 4.19. As linhas 4 a 8 especializam o *trait* para o tipo de dado `int`, definindo variáveis como o valor máximo e mínimo de um inteiro e se int é `unsigned` ou não. As linhas 9 a 13 fazem o mesmo para `unsigned` e as linhas 14 a 18 fazem o mesmo para `signed char`. As linhas 23 a 33 definem uma função para calcular o menor valor de uma coleção. A linha 24 inspeciona o tipo de dado armazenado no iterador. A linha 25 utiliza o trait especializado para este tipo de dado genérico para inicializar a variável com o tamanho máximo, mantendo o código genérico.

Exemplo 4.19 – Definição de uma meta função *traits* para tipos de dados numéricos.

```
1  template<typename T>
2  struct numeric_traits;
3
4  template<> struct numeric_traits<int> {
5    static constexpr bool is_unsigned = false;
6    static constexpr int max_value = INT_MAX;
7    static constexpr int min_value = INT_MIN;
8  };
9  template<> struct numeric_traits<unsigned> {
10   static constexpr bool is_unsigned = true;
11   static constexpr unsigned max_value = UINT_MAX;
12   static constexpr unsigned min_value = 0;
13 };
14 template<> struct numeric_traits<signed char> {
15   static constexpr bool is_unsigned = false;
16   static constexpr signed char max_value = CHAR_MAX;
17   static constexpr signed char min_value = CHAR_MIN;
18 };
19
20 ...
21
22 template<typename It, typename Sentinel>
23 auto min(It begin, Sentinel end) {
24   using value_type = decltype(*begin);
25   value_type return_value = numeric_traits<value_type>::max_value;
26   while( begin != end ) {
27     if( *begin < return_value ) {
28       return_value = *begin;
29     }
30     ++begin;
31   }
32   return return_value;
33 }
```

4.3.4 *Type erasure*

Polimorfismo é uma técnica muito utilizada em programação quando se tem uma operação que é realizada de jeitos diferentes por tipos de dados diferentes e estes não

importam muito na hora de chamar a função. Em C++, o domínio de polimorfismo é separado em duas frentes:

- Polimorfismo em tempo de compilação, onde o tipo de dado é passado como um *template* para a função ou classe e o compilador cria instâncias diferentes para cada definição. Isso faz com que o desempenho seja máximo, sem custo extra no tempo de execução mas aumentando o tamanho do código executável devido à instanciação de diferentes funções muito parecidas. Além disso, o tipo de dado é decidido em tempo de compilação, não tendo a flexibilidade de alternar entre diferentes tipos durante a execução do programa.

- Polimorfismo em tempo de execução, onde usualmente é declarada uma interface em uma superclasse e os tipos de dado que desejam flexibilizar uma chamada de função devem herdar dessa superclasse. Diferente do polimorfismo em tempo de compilação, a necessidade de passar por um ponteiro para uma *virtual table* deixa o código mais lento, assim como a dificuldade do compilador aplicar otimizações do tipo *inline* às chamadas de função, que abrem espaços a outras possíveis otimizações. Porém, a flexibilidade dos possíveis tipos de dados aumenta, podendo instanciar diferentes tipos de dado a uma variável.

O seguinte parágrafo elenca técnicas clássicas utilizando herança para polimorfismo em tempo de execução e possíveis problemas encontrados nestas técnicas.

Implementações clássicas de polimorfismo em tempo de execução têm uma superclasse que define métodos virtuais abstratos que devem ser implementados por diferentes subclasses. Desta forma, em cada ponto que se deseja uma operação polimórfica, é definido um parâmetro de ponteiro ou referência para uma instância da superclasse e instâncias de subclasses são passadas como argumentos para dentro da função, invocando os métodos de forma polimórfica. Dito isto, pode-se observar alguns pontos limitantes sobre esta técnica:

1. Cada tipo diferente de dado que se deseja utilizar na função deve herdar da superclasse que define a interface. Isso não é um problema caso se tenha controle total de todos os tipos de dados utilizados, mas impossibilita integração com bibliotecas externas, tipos primitivos e causa problemas de herança múltipla e longas hierarquias.

2. Como o uso da superclasse requer ponteiros e referências, perde-se semântica de valor [1] dos dados. Uma cópia do dado apenas copia o ponteiro ou referência àquele dado, sendo necessária a definição de métodos como `clone()` ou `copy()`, assim como alocação dinâmica de memória.

[1] Semântica de valor se comporta como tipos primitivos. Uma cópia do dado copia o valor, não a referência.

Type erasure é uma técnica de implementação que mistura a flexibilidade do polimorfismo em tempo de execução com a não intrusão e a semântica de valor presentes no polimorfismo em tempo de compilação e nos tipos primitivos da linguagem. Para tanto, é definida uma interface e dois ponteiros: um ponteiro de tipo `void` para o dado em si e um ponteiro para uma *virtual table* que, para cada função definida na `struct` da tabela, recebe como primeiro parâmetro o ponteiro para o dado e depois os parâmetros de cada uma das funções necessárias em si. Dentro da interface, então, é definido um método de *assignment* recebendo um *template* T para um dado que seja conformante à interface e dentro deste método é criada a *virtual table*, usando lambdas que decaem a ponteiros para funções que apenas dão cast no ponteiro `void` do primeiro parâmetro para o tipo de dado `T` e chamam o método concreto. Desta forma, o conceito de *virtual table* se separa do conceito de herança, com o mecanismo de *template* se encarregando da criação da mesma. Uma definição de uma classe com *type erasure* pode ser vista no exemplo 4.20. Esta classe se preocupa apenas em armazenar e destruir qualquer tipo de dado. Nas linhas 1 a 4, temos a definição da *virtual table*. O primeiro parâmetro passado para qualquer um dos métodos da *virtual table* é do tipo ponteiro `void` e corresponde ao objeto atualmente armazenado. As linhas 32 a 43 tratam da construção da *virtual table* para um tipo de dado T. Primeiro, na linha 34 é definida uma variável estática, de forma que o endereço de memória da *virtual table* seja o mesmo para todos os objetos do mesmo tipo. Então, para cada um dos ponteiros para funções, o primeiro parâmetro recebe um cast para o tipo T e depois é despachado para o método correto. As funções são geradas em tempo de compilação através do uso de lambdas. Um exemplo de função de cópia pode ser visto nas linhas 35 a 39 e de um destrutor na linha 40. As linhas 7 a 31 definem a classe que realiza o *type erasure* em si. As linhas 11 a 15 utilizam SFINAE, técnica vista na seção 4.3.1 para restringir os tipos possíveis a todos os tipos diferentes de `any`, de forma a não obstruir o *copy constructor* e constroem um objeto na *heap*, armazenando-o num ponteiro void e construindo a *virtual table* para `T`. A linha 29 define um *copy constructor*, que apenas utiliza a *virtual table* de outro *any* para copiar o valor e a *virtual table* para o novo objeto. A linha 30 invoca o destrutor do objeto atualmente armazenado através da *virtual table*. As linhas 16 a 28 definem métodos de copy assignment entre *any* e tipos genéricos.

Para o armazenamento de dado desta técnica, um *small buffer optimization*, como visto na sessão 4.3.2, pode ser utilizado para objetos de até um certo tamanho, evitando alocações dinâmicas de memória. Outra possibilidade é o não armazenamento do dado, guardando apenas uma *view* no ponteiro `void` e não se preocupando com a memória em si.

Exemplo 4.20 – Exemplo type erasure

```
1  struct v_table_ {
2    void ( *copy ) ( void*, void* );
3    void ( *destroy ) ( void* );
4  };
5  template<typename T>
6  v_table_* get_v_table_for();
7  class any {
8    void* m_obj;
9    v_table_* m_v_table;
10 public:
11   template<typename T, enable_if_t<!is_same_v<remove_reference<T<, any>>>
12   any( T&& obj ) {
13     m_obj = ( void* )( new T( forward<T>( obj ) ) );
14     m_v_table = get_v_table_for<T>();
15   }
16   template<typename T, enable_if_t<!is_same_v<remove_reference<T>, any>>>
17   any& operator=( T&& obj ) {
18     m_v_table->destroy( m_obj );
19     m_obj = ( void* )( new T( forward<T>( obj ) ) );
20     m_v_table = get_v_table_for<T>();
21   }
22   any& operator=( const any& rhs ) {
23     if( &rhs != this ) {
24       m_v_table->destroy( m_obj );
25       rhs.m_v_table->copy( rhs.m_obj, this );
26     }
27     return *this;
28   }
29   any( const any& rhs ) { rhs.m_v_table->copy( rhs.m_obj, this ); }
30   ~any() { m_v_table->destroy( m_obj ); }
31 };
32 template<typename T>
33 v_table_* get_v_table_for() {
34   static v_table_ _ = {
35     []( void* src, void* dst ) {
36       any* other = static_cast<any*>( dst );
37       other->m_v_table = get_v_table_for<T>();
38       other->m_obj = (void*)( new T( *( T* )( src ) ) );
39     },
40     []( void* obj ) { T* actual = ( T* ) obj; delete actual; }
41   };
42   return &_;
43 }
```

4.3.5 Alocador de memória

O alocador de memória utiliza técnicas de *type erasure* definidas na seção 4.3.4 para criar uma *view* em tempo de compilação para qualquer tipo de dado que contenha funções de alocação e desalocação de memória. As seguintes seções tratam sobre técnicas utilizadas na implementação da *view*.

4.3.5.1 Criação das funções de alocação e desalocação

Para a criação das funções de alocação e desalocação de memória, são utilizadas técnicas SFINAE para detectar métodos. As funções de alocação e desalocação recebem um tipo genérico T equivalente ao alocador e usam if constexpr para testar qual das funções deve ser chamada. Isso otimiza o código, descartando todos os ramos que o dado não suporta a operação. Essas funções ainda recebem parâmetros de alinhamento, no caso da alocação e tamanho, no caso de desalocação. Caso o dado não suporte esses parâmetros nas suas funções, esses parâmetros não são utilizados e o compilador pode na sua etapa de otimização, facilmente apenas removê-los.

4.3.5.2 Criação da *virtual table*

A *virtual table* contém métodos que recebem um `void *` como primeiro parâmetro e a implementação mais geral de dados, com alocações alinhadas e desalocações que recebem o tamanho do dado. A implementação da *virtual table* então retorna uma estrutura com funções lambda, realizando *casts* para o tipo de dado específico e criando as funções como definido na seção 4.3.5.1, passando todos os parâmetros para a função. Cada *virtual table* contém apenas 2 métodos, `allocate` e `deallocate` e estas são compartilhadas entre todas as instâncias do mesmo tipo de dado.

4.3.5.3 *View* para um alocador de memória

Uma *view* para um alocador de memória guarda uma referência em um *void ** e uma *virtual table* que realiza as operações do tipo de dado em si. Como a *view* não é responsável pela memória do alocador, essa classe é relativamente simples, fazendo apenas a ponte entre um *template* em tempo de compilação e um *void ** em tempo de execução. O único detalhe de implementação a ser observado é na diferenciação entre um construtor recebendo um objeto T genérico e o *copy constructor*. Devido a regras da linguagem C++, mesmo que o tipo de dado recebido no construtor seja o mesmo tipo da classe, um construtor que recebe `T&&` é preferido a um construtor que recebe `const allocator&`. Para resolver isso, o *framework* utiliza SFINAE para escolher a especialização certa.

4.3.5.4 Alocador de memória padrão

O *framework* armazena um *void ** e uma *virtual table* globais como alocador de memória padrão. Ao chamar a função de usar o alocador padrão, uma *view* usando essas 2 variáveis é criada e retornada usando um construtor privado da classe `allocator`. Para tanto, essa função é marcada como friend. Um *overload set* da função `setdefaultallocator` provê um método para um tipo de dado genérico, que cria uma *virtual table* e a armazena junto com a referência ao dado nas variáveis globais e um método para um `allocator`, armazenando a *view* contida por este nas variáveis. Este alocador tem como seu valor inicial uma *struct* que apenas chama as funções malloc e free.

4.3.6 Árvore K-D implementada em arranjo

Uma árvore K-D pode ser implementada eficientemente num arranjo compacto, em que cada elemento do arranjo representa um dado inserido na árvore. Desta forma, a árvore possui uma melhor localidade de cache e fica compacta na memória. As seções a seguir tratam das diferentes otimizações e técnicas de implementação aplicadas para essa estrutura.

4.3.6.1 Organização dos dados

Como otimização inicial, pode-se notar que, para cada nó, não é necessário guardar nem ponteiros para os nós filhos nem a dimensão atual se, dado a posição de memória de um nó, for possível calcular a posição de memória de ambos os nós filhos. No caso, percebe-se que, para cada elemento, tem-se a divisão do espaço em duas partes iguais, caso o número de elementos processados pelo nó seja ímpar, ou em duas partes com diferença de tamanho um, caso o número seja par. Define-se um nó da árvore como uma dupla (I, T), onde I é o índice do nó no *array* e T é o número de elementos presentes na subárvore. Assim, podemos encontrar os filhos de um determinado nó a partir da fórmula

$$FE \leftarrow \begin{cases} (I - \frac{T}{4} - 1, \frac{T}{2}), & \text{se tamanho bloco é ímpar} \\ (I - \frac{T}{4}, \frac{T}{2}), & \text{caso contrário} \end{cases} \tag{4.1}$$

$$FD \leftarrow \begin{cases} (I + \frac{T}{4} + 1, \frac{T}{2}), & \text{se tamanho bloco é ímpar} \\ (I + \frac{T}{4}, \frac{T}{2} - 1), & \text{caso contrário} \end{cases} \tag{4.2}$$

onde FE é o filho da esquerda, FD é o filho da direita e o nó raiz é igual a ($\frac{T}{2}$, T), sendo T o número de elementos da árvore. Outra observação é a não necessidade de armazenar o tamanho dos blocos, sendo possível calcular estes em tempo de execução. O caso base ocorre quando o tamanho do bloco é 1, sendo este um nó folha sem nenhum filho na esquerda ou direita. Isso causa um *trade-off* de consumo de memória por ciclos de CPU,

que tende a otimizar mais o programa devido às CPUs de hoje em dia serem muito mais rápidas que acesso à memória. Para exemplificar essa fórmula, observa-se a figura 2 como uma árvore KD de tamanho 10. Dado um *array* na figura com a raiz da árvore em destaque. Cada um dos diferentes tons de cinza demonstra um diferente nível na árvore, com a raiz sendo o tom mais escuro e os níveis mais abaixo os tons mais claros. Para cada posição no *array*, tem-se setas para a esquerda e para a direita apontando outros índices de acordo com a aplicação das fórmulas definidas nas equações 4.1 e 4.2. Por fim, as setas horizontais representam o tamanho de bloco T de cada nó, onde as setas mais abaixo representam níveis da árvore mais baixos. Blocos de tamanho 1 têm seu tamanho omitido para não poluir a imagem.

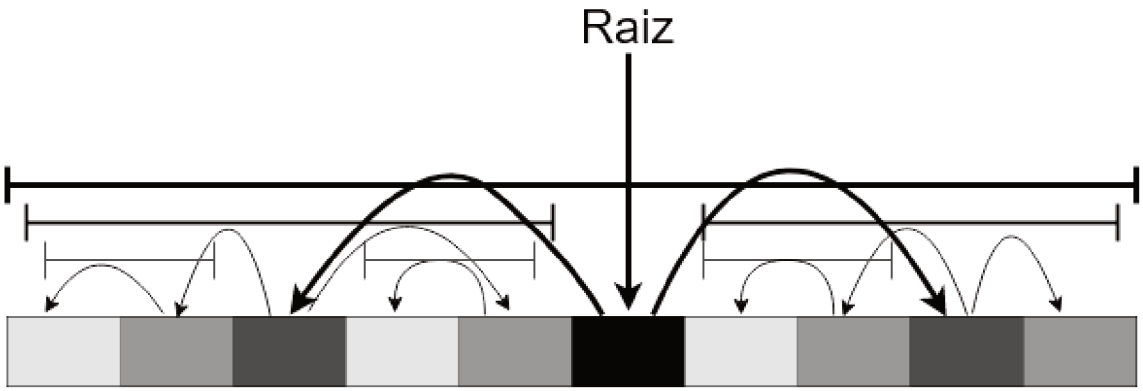

Figura 2 – Exemplo de uma árvore KD de tamanho 10 em *array*.

4.3.6.2 K vizinhos mais próximos

Para o cálculo dos K vizinhos mais próximos, é usada uma estrutura *heap* de forma que seja possível remover rapidamente o K elemento mais próximo no caso de um outro elemento estar mais próximo. De forma a otimizar a busca, um vetor utilizando *small buffer optimization* de até 10 elementos é usado para realizar a busca, para os casos de K ser um valor pequeno. Desta forma, o código aloca memória na *heap* apenas caso K seja maior que 10.

4.3.6.3 Recursão

A recursão de um nível K para um nível $K+1$ é otimizada em tempo de compilação para não necessitar a passagem do nível atual nem da dimensão atual para a chamada de função. O *framework* utiliza de *template metaprogramming* para chamar o próximo nível da recursão com a dimensão certa. Isso otimiza o código das chamadas recursivas, removendo a necessidade de *branches* em troca do número de dimensões do dado ser necessário em tempo de compilação.

Cada chamada recursiva recebe como *template* um inteiro equivalente a dimensão atual. Para o nível de recursão $K+1$, a dimensão pode ser calculada como $D(K+1) = D(K)+1 \mod Dim(dado)$, onde $Dim(dado)$ é o número de dimensões do tipo de dado definido nos traits `dimensional_traits`. A raiz sempre recebe a dimensão 0. A figura 3 exemplifica tais chamadas, onde cada um dos números entre <> representa a dimensão atual passada como *template* e calculada em tempo de compilação para um dado de duas dimensões. Exemplificando um dado de duas dimensões, um ponto $P = (x, y)$ tem a primeira dimensão como x e a segunda dimensão como y, $D < 0 > (P) = x$ e $D < 1 > (P) = y$, onde P é o tipo de dado.

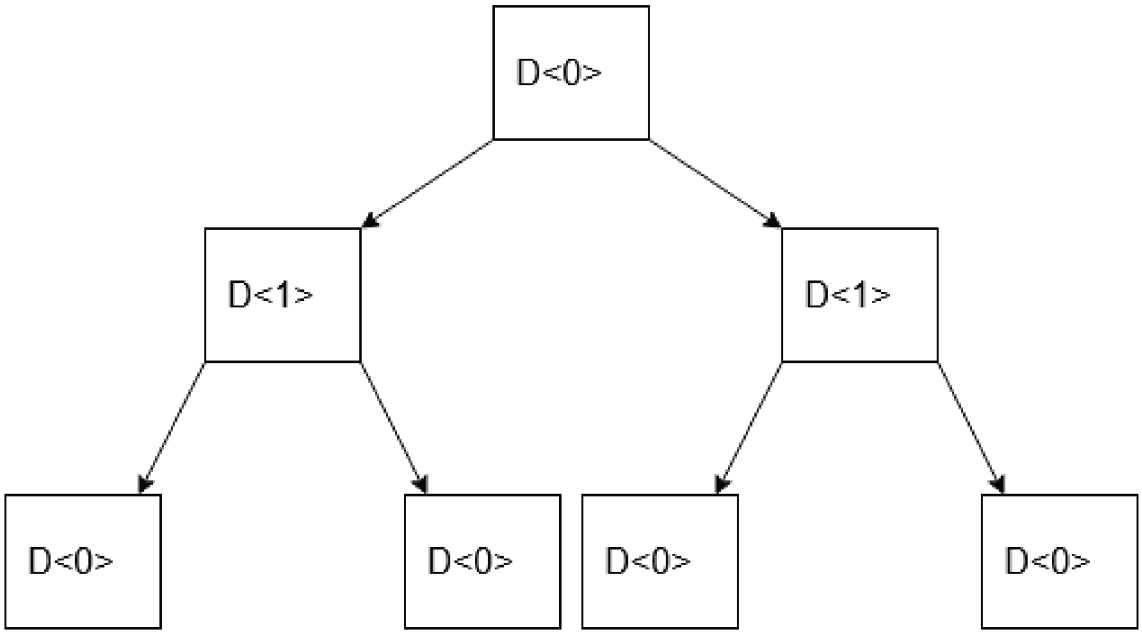

Figura 3 – Exemplo de chamadas recursivas usando *template* para dado de 2 dimensões.

4.3.7 *Quadtree*

A árvore *quadtree* é implementada usando 3 diferentes vetores com uma referência ao mesmo alocador de memória. As seções a seguir tratam sobre as diferentes técnicas utilizadas, com o esquema da árvore na memória e técnicas de programação genérica.

4.3.7.1 Armazenamento dos dados

A árvore é implementada em 3 diferentes vetores conforme a figura 4. Os 3 vetores utilizam o mesmo alocador de memória, passado ao construtor da *quadtree*. Para redução de consumo de memória, os *bounding rects* dos nós da árvore são calculados em tempo de execução, tendo apenas o do nó raiz armazenado. Este *bounding rect* é passado como argumento para o construtor e armazenado na árvore. Os retângulos dos subsequentes nós são calculados dividindo o retângulo do nó pai em 4 partes, com o primeiro filho

utilizando o retângulo superior esquerdo, o segundo o retângulo superior direito, o terceiro o retângulo inferior esquerdo e o quarto o retângulo inferior direito.

O primeiro vetor armazena os nós da árvore. O elemento 0 do vetor representa o nó raiz. Cada um destes nós pode armazenar 2 diferentes tipos de valores: um índice para o primeiro nó filho, no caso de um nó intermediário ou um índice para o primeiro elemento, no caso de um nó folha. Cada nó da árvore utiliza apenas um inteiro de tamanho 32 bits. Para diferenciar um nó intermediário de um nó folha, é utilizado o bit mais significativo. Os 31 bits restantes endereçam o elemento contido pelo nó. No caso de um nó intermediário, dado um inteiro de 31 bits I, o *offset* dos 4 filhos no vetor são dados por I, I + 1, I +2 e I + 3. No caso de um nó folha, o valor especial -1 é utilizado para indicar um nó folha vazio. Isso pode ser observado na figura 4. O primeiro elemento representa o nó raiz, um nó intermediário que aponta para o índice 1. Os próximos 3 filhos são encontrados de forma contígua na memória. Os 3 filhos então são nós folhas, apontando para índices no vetor de elementos. O nó no índice 4 contém o valor especial -1, indicando um nó folha vazio e apontando para o elemento vazio.

O segundo vetor é implementado como listas encadeadas de índices para os objetos armazenados em si. A implementação é feita desta maneira pois o mesmo objeto pode ser encontrado em diversos nós folha, caso o seu *bounding rect* intersecte mais de uma folha. Cada elemento armazena 2 índices de 32 bits: o primeiro representa o *offset* para o próximo elemento da lista encadeada, com o valor especial -1 representando o final da lista e o segundo o *offset* ao vetor de elementos em si. Na figura, cada lista encadeada é representada por um tom diferente. O tamanho máximo de cada lista antes do nó precisar se subdividir em 4 é dado por um parâmetro de *template* passado como *trait* à estrutura, com este tamanho sendo infinito caso o nó não consiga mais se dividir ou um valor de profundidade também passado como argumento de *template* seja atingido. O elemento vazio é apenas simbólico, não sendo necessário armazená-lo.

Por fim, o último vetor contém os objetos armazenados em si, junto com seus *bounding rect*s.

4.3.7.2 Dados suportados

Uma estrutura *quadtree* suporta dados que definam uma conversão para um `geometricks::rectangle`, conforme descrito na seção 4.2.2.4. Os retângulos armazenados pela estrutura são todos definidos por `int32_t`, de forma que a conversão para este tipo deve ser definido pelo usuário. No caso da necessidade de inserção de outro objeto geométrico, é inserido o *bounding rect* deste objeto para facilitar os cálculos internos e aumentar o desempenho.

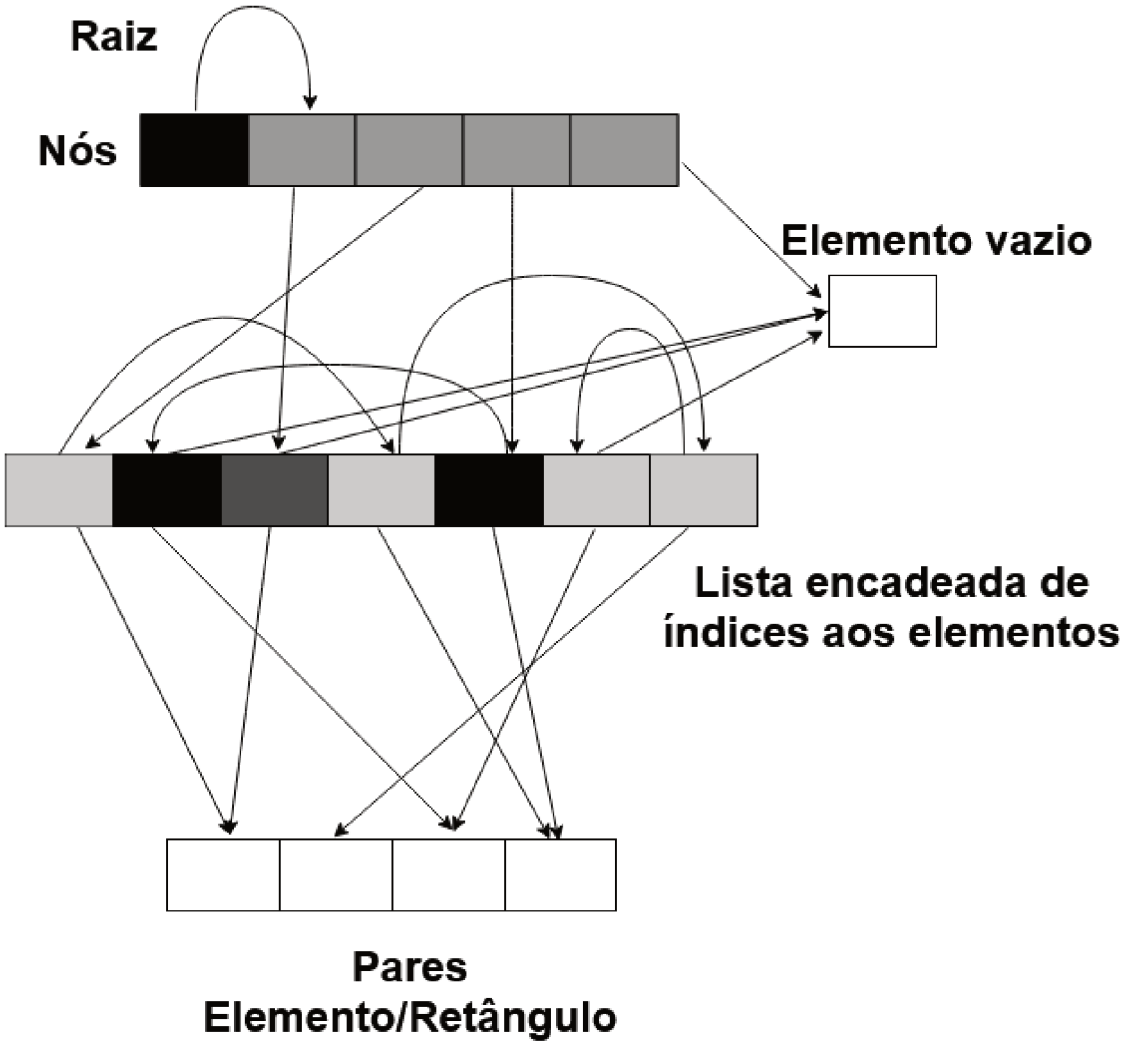

Figura 4 – Exemplo de uma estrutura de uma *quadtree* com 4 elementos.

4.3.8 *Octree*

A implementação de *octree* é muito parecida com a implementação de *quadtree*, usando as mesmas estruturas descritas na seção 4.3.7.1. Como diferenças, são apontadas o primeiro vetor da figura 4 tendo 8 filhos ao invés de 4, mas apenas contendo um índice ao primeiro assim como no caso da *quadtree*. Como os objetos armazenados são 3D, os pares do último vetor são de Elemento/Paralelepípedos e as funções auxiliares usadas para a construção da *octree* retornam paralelepípedos no lugar de retângulos, discutido na seção 4.2.2.5. Não há diferenças no resto da implementação significantes a serem mencionadas.

5 Resultados e discussões

Como principal resultado obtido deste trabalho, tem-se o código fonte disponível em <https://github.com/mitthy/TCC>, onde o *framework* sugerido é implementado na linguagem C++.

O *framework* tem como principal objetivo a alta personalização das estruturas. Isso pode ser feito de duas maneiras diferentes: de forma polimórfica dinâmica ou utilizando o mecanismo de *template* da linguagem C++. Com exceção do módulo de memória, o *framework* opta pela utilização dos mecanismos estáticos, devido a forma dinâmica ser intrusiva, com os dados precisando herdar de tipos definidos internamente. Porém, devido ao mecanismo de *template* criar uma cópia da função ou classe para cada definição, isso causa um *tradeoff* no tempo de compilação e tamanho do executável. O módulo de memória utiliza de um polimorfismo *ad-hoc* com técnicas de *type erasure* para separar o tipo de alocador de memória dos tipos de estrutura, podendo tratar estruturas do mesmo tipo de dado mas que recebem alocadores diferentes de forma uniforme às custas de desempenho no tempo de execução.

Enquanto outros trabalhos na área focam mais em indexações com *r-trees*, aproximações de *nearest neighbor* e programações de jogos, o *framework* implementado tem como principal objetivo documentar as diferentes técnicas de implementação utilizadas na criação de estruturas de dados genéricas. Isso é explorado mais a fundo na seção 4, onde cada uma das diferentes técnicas de implementação utilizadas são exploradas mais a fundo. Além disso, o *framework* é altamente personalizável e não intrusivo, se assemelhando muito ao Boost.Geometry neste aspecto. Outros trabalhos como FLANN alcançam desempenho melhor, mas são intrusivos aos dados ou apenas retornam dados aproximados com um *trade-off* de sua eficiência.

A estrutura de árvore KD é implementada como um arranjo compacto de forma a salvar memória, conforme visto na seção 4.3.6. Esta representação é possível devido ao fato do número de elementos da árvore ser constante e a divisão ser sempre feita exatamente na metade. A árvore utiliza vetores alocados na *stack* até um certo número de elemento internamente quando possível para acelerar operações no caso médio, e permite alocadores de memória personalizados. Foram testadas ainda outras alternativas de implementação, utilizando ponteiros para os filhos de cada nó e representando a árvore na forma de uma *heap*. A técnica utilizando ponteiros utilizou mais memória para guardar cada nó, tendo um *overhead* extra do tamanho de 2 ponteiros por elemento, e não foi eficiente com a cache, devido a alocação dos blocos não ser contígua, sendo então descartada. A técnica utilizando uma representação de *heap* gastou muito mais memória que precisava. Outras

considerações tomadas à árvore dizem respeito ao armazenamento de chaves e valores, possibilitando o uso desta como um mapa. Pelo levantamento, esta estrutura é utilizada para dados de não muitas dimensões, em aplicações onde não acontecem muitas inserções dinâmicas e sim bastantes operações de busca.

Para a implementação de *quadtree* e *octree*, foi escolhido apenas manter índices aos elementos nos nós e guardá-los em uma lista de vetor encadeada. Isso se dá devido ao fato de que guardar um vetor de elementos nos nós é muito mais custoso em memória, precisando guardar metadados dos elementos em cada um dos nós internos. Guardando apenas um índice, é possível diminuir o uso de memória de cada um dos nós para apenas 32 bits. Outra consideração que poderia aumentar um pouco o uso de memória mas acelerar os algoritmos seria armazenar informações tais como o número de elementos de cada nó, não precisando calcular dinamicamente este valor. A árvore ainda permite a inserção de diferentes elementos no mesmo ponto, permitindo mais flexibilização, mas com perda de desempenho e necessitando cuidar de casos de infinitas subdivisões. Para isso, são usados valores externos indicando a profundidade máxima e o número máximo de elementos por nó. Cada nó guarda apenas um índice ao seu primeiro filho, devido ao fato da divisão de espaço sempre ocorrer de 4 em 4 filhos, é possível achar os próximos filhos dinamicamente sem muitos custos e diminuir o gasto de memória. Essa representação torna possível identificar se um nó é folha ou não utilizando apenas 1 bit, enquanto os outros 31 bits são utilizados para indexar. Outra consideração feita foi de utilizar a árvore como um mapeamento de retângulos para objetos. Como a árvore *octree* é muito parecida com a *quadtree*, as técnicas e considerações de implementação são as mesmas. Essas estruturas permitem uma inserção e remoção eficiente, com o levantamento mostrando que são utilizadas em aplicações que necessitam de criação, atualização e busca de diversos objetos, como jogos que têm milhares de partículas na tela e precisam de otimizações para verificação de colisão de objetos, assim como para representar regiões de espaço que contenham uniformemente o mesmo valor. Essa última aplicação não é suportada pela implementação realizada no *framework*.

A interface utiliza de objetos ponto de personalização para as operações personalizáveis, *views* para alocadores de memórias e funções membro dentro das classes das estruturas. Uma possível consideração é extrair algumas funções de busca como funções livre, criando um *overload set* para as estruturas em geral. Os pontos de personalização então verificam chamadas de acordo com o protocolo e não sofrem *override* em si, evitando a necessidade do uso de using como encontrado em swap. A interface das estruturas em si possuem a parte estática de *template* e a parte das funções. Enquanto *templates* são passados como argumento para o tipo de dado em si, permitindo a personalização de operações básicas em tempo de compilação, as funções aceitam quando possível objetos *callable* para flexibilizar as operações.

6 Conclusão

O desenvolvimento deste trabalho disponibilizou uma biblioteca *open source* para estruturas de dados geométricas na linguagem C++. Um levantamento inicial de quais estruturas e algoritmos a serem implementados levou a decisão de implementação de*KD trees*, *quadtrees*, *octrees* e *rtrees*, devido a essas estruturas serem mais clássicas na literatura e mais simples de entender a implementação, o que facilita no objetivo de documentar de forma didática o documento, focando mais nas técnicas utilizadas e não em detalhes das estruturas.

Um estudo de cada uma destas estruturas foi feito, com diferentes técnicas de implementação revisadas e ponderadas para cada uma delas, assim como uma revisão em outros *frameworks* similares para este tipo de estrutura. Depois de tomadas as decisões de implementações, paralelamente com a criação deste documento, as estruturas *kd tree* e *quadtree* foram implementadas na linguagem C++, assim como um *framework* de suporte para personalização de dados e gerenciamento de memória, possibilitando ao usuário definir diferentes padrões de acesso. Outras estruturas de baixo nível são implementadas internamente para otimizações de algoritmos, operações e para fazer uma ponte entre diferentes componentes do *framework*. Um conjunto de testes utilizando Google Test ainda é implementado para fins de testes de corretude e usabilidade de cada uma das implementações. A documentação da implementação de forma didática então é feita neste documento, com as diferentes técnicas utilizadas na implementação do *framework* descritas na seção 4.

Para trabalhos futuros, tem-se a implementação das estruturas *octree*, *rtree* e *grid*, assim como a possibilidade de utilizar as estruturas atualmente implementadas como um mapa, melhorando a usabilidade do *framework*. Outras sugestões de trabalhos futuros incluem a inclusão de implementações de outras métricas de distância, *benchmarks* com outras implementações, implementação de tipos concretos de alocadores de memória, otimizações de baixo nível nas estruturas existentes, inclusão de novas formas geométricas e otimização de parâmetros estáticos para estruturas de baixo e alto nível definidos no código.

Referências

APPLE GameplayKit. Disponível em: <https://developer.apple.com/documentation/gameplaykit>. Acesso em: 23.11.2019. Citado na página 37.

BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM*, ACM, New York, NY, USA, v. 18, n. 9, p. 509–517, set. 1975. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/361002.361007>. Citado na página 20.

BENTLEY, J. L.; HAKEN, D.; SAXE, J. B. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, ACM, v. 12, n. 3, p. 36–44, 1980. Citado na página 22.

BLUM, M. et al. Time bounds for selection. *J. Comput. Syst. Sci.*, v. 7, n. 4, p. 448–461, 1973. Citado na página 22.

BOOST Software License. 2003. Disponível em: <https://www.boost.org/LICENSE_1_0.txt>. Acesso em: 19.11.2019. Citado na página 35.

BOOST.GEOMETRY. Disponível em: <https://www.boost.org/doc/libs/1_71_0/libs/geometry/doc/html/index.html>. Acesso em: 19.11.2019. Citado 2 vezes nas páginas 18 e 35.

BOXTREE. Disponível em: <https://documen.tician.de/boxtree/>. Acesso em: 23.11.2019. Citado na página 37.

BSD License. Disponível em: <https://opensource.org/licenses/BSD-3-Clause>. Acesso em: 19.11.2019. Citado na página 37.

CPP Reference Trivially Copyable. Disponível em: <https://en.cppreference.com/w/cpp/types/is_trivially_copyable>. Acesso em: 09.09.2020. Citado na página 63.

ERICSON, C. *Real-time collision detection*. [S.l.]: CRC Press, 2004. Citado na página 26.

FLANN - Fast Library for Approximate Nearest Neighbors. Disponível em: <https://github.com/mariusmuja/flann>. Acesso em: 19.11.2019. Citado 2 vezes nas páginas 17 e 37.

GAMMA, E. et al. *Design Patterns*. [S.l.]: Addison-Wesley, 1994. ISBN 0-201-63361-2. Citado na página 37.

HOARE, C. A. Algorithm 65: find. *Communications of the ACM*, ACM, v. 4, n. 7, p. 321–322, 1961. Citado na página 22.

LLVM. Disponível em: <https://llvm.org/>. Acesso em: 22.11.2019. Citado na página 65.

MEAGHER, D. Geometric modeling using octree encoding. *Computer graphics and image processing*, Elsevier, v. 19, n. 2, p. 129–147, 1982. Citado na página 32.

OPENCL. Disponível em: <https://www.khronos.org/opencl/>. Acesso em: 23.11.2019. Citado na página 37.

ROUSSOPOULOS, N.; KELLEY, S.; VINCENT, F. Nearest neighbor queries. In: *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. [S.l.: s.n.], 1995. p. 71–79. Citado na página 23.

SAMET, H. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 16, n. 2, p. 187–260, 1984. Citado na página 25.

SARWAR, B. et al. Item-based collaborative filtering recommendation algorithms. In: *Proceedings of the 10th international conference on World Wide Web*. [S.l.: s.n.], 2001. p. 285–295. Citado na página 23.

VANDEVOORDE, D.; JOSUTTIS, N. M. *C++ Templates: The Complete Guide*. [S.l.]: Addison-Wesley Professional, 2003. ISBN 0-201-73484-2. Citado na página 63.

WHANG, K.-Y. et al. Octree-r: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, IEEE, v. 1, n. 4, p. 343–349, 1995. Citado na página 32.

Glossário

ADL Funcionalidade da linguagem C++ que inclue o *namespace* dos tipos de dado na busca por uma chamada de função num *overload set*.

aligned_alloc Função padrão da biblioteca C similar a malloc que realiza alocação alinhada dinâmica de memória.

aligned_storage `Struct` da linguagem C++ que armazena um *buffer* na *stack*, com seu tamanho e alinhamento de memória definidos em parâmetros de *template*.

auto Palavra reservada da linguagem C++ utilizada para dedução automática de um tipo de dado ou retorno de função.

bounding rect Retângulo que engloba um objeto geométrico.

bulk loading Operação de inicializar uma estrutura de dados com um conjunto de entrada, possibilitando otimizações de acordo com características pertinentes a entrada.

callable Qualquer objeto ou função da linguagem C++ que pode ser invocado como um método, utilizando o operador ().

cast Operação explícita para tratar um tipo de dado como outro da linguagem C++.

constexpr Marcação de uma função que pode ser calculada em tempo de compilação, porém com um uso limitado da linguagem. Também pode ser usado para marcar variáveis a serem calculadas em tempo de compilação.

copy constructor Construtor de uma classe `T` que recebe um parâmetro do tipo `const T&` seguido de uma lista de parâmetros opcionais com valores pré-definidos e cria uma cópia do objeto.

false_type `Struct` da linguagem C++ que define um tipo de dado para a constante booleana `false`.

fold expression Funcionalidade implementada em C++17 que permite operações de operadores como `"+"`, `"-"`, `","`, `"/"`, etc. em uma lista de variadic templates.

free Função padrão da biblioteca C que realiza desalocação dinâmica de memória alocada pela função malloc.

friend Declaração de uma função ou classe que tem acesso aos atributos privados do tipo de dado.

if constexpr *Branch* realizado em tempo de compilação que, dado um valor booleano calculado também em tempo de compilação, condicionalmente compila um código.

iter_swap Função da biblioteca std lib do C++ que recebe 2 iteradores e troca o valor contido por eles.

lambda `Struct` anônimas em C++ que definem um operador() e podem usar usadas como funções.

malloc Função padrão da biblioteca C que realiza alocação dinâmica de memória.

memcpy Função da biblioteca C otimizada que copia um vetor de *bytes* para uma outra região distinta de memória. Necessita que o tipo de dado seja trivially_copyable para ser usada.

namespace Palavra reservada da linguagem C++ que separa um conjunto de funções, tipos de dados e variáveis em um escopo separado com o intuito de evitar choques em projetos grandes.

overload set Conjunto de chamadas de uma mesma função com o mesmo nome que recebem argumentos de tipos de dados diferentes.

override Redefinição de um método que recebe um tipo de dado diferente para chamadas polimórficas em tempo de compilação ou execução.

placement new Operador em C++ que recebe um ponteiro de memória para uma localização onde se deseja construir um objeto e o constrói neste lugar.

policy trait Parâmetro de *template* passado a certas classes de forma a mudar seu comportamento.

std lib Biblioteca da linguagem C++ que define estruturas de dados e algoritmos genéricos.

sum type Tipo de dado que armazena apenas um entre vários tipos de dados declarados. Normalmente tem seu tamanho e alinhamento definidos pelo máximo entre os diferentes tipos de dado possíveis de se representar.

swap Função da biblioteca do C++ que troca os valores contidos por 2 objetos.

tag type Tipo de dado que é passado como argumento a funções para desambiguar um *overload set*, não armazenando nenhum dado.

template Mecanismo da linguagem C++ que recebe um tipo ou uma constante em tempo de compilação como parâmetro e cria diferentes instâncias de funções ou classes.

template metaprogramming Programação em tempo de compilação utilizando *templates* para produzir tipos de dado.

trivially_copyable *Trait* definido na std lib que retorna se um tipo de dado pode ser copiado trivialmente, ou seja: se o tipo não possui um *copy constructor* e pode ser trivialmente copiado com memcpy.

trivially_destructible *Trait* definido na std lib que retorna se um tipo de dado pode ser destruído trivialmente, ou seja: se o tipo não possui um destrutor definido pelo usuário e é apenas um no-op.

true_type `Struct` da linguagem C++ que define um tipo de dado para a constante booleana `true`.

union *sum type* padrão da linguagem C. Uma instância da `union` pode ser qualquer um entre os tipos definidos internamente, mas apenas um ao mesmo tempo.

using Palavra reservada da linguagem C++ que, ou define um alias para outro tipo de dado, ou importa uma função ADL para um *overload set*. Muito utilizado com swap.

variadic template Lista de tamanho ilimitado de *templates* passados como argumentos a uma estrutura ou função.

view Tipo de dado que se comporta como uma referência, ou seja, não cuida da memória de um objeto e apenas disponibiliza acesso aos dados.

viewport Descreve as coordenadas de um mundo. Podem ser realizadas transformadas de uma *viewport* para outra, passando as coordenadas de um objeto de um sistema para outra.

virtual table Tabela de ponteiros para funções usada em polimorfismo, de forma que instâncias diferentes chamem funções diferentes. Em uma implementação padrão usando herança, cada subclasse possui sua própria tabela.

void_t Tipo de dado que mapeia uma lista ilimitada de tipos para o tipo `void`. Muito utilizada em SFINAE.

Apêndices

APÊNDICE A – Artigo SBC

Geometricks: Um framework de estruturas de dados geométricas em C++

Lucas F. Roman, Maicon R. Zatelli

[1]Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 88040-900 – Florianópolis – SC – Brazil

Abstract. *Geometric objects, such as points, lines, polygons and cubes, are not intrinsically ordered in respect to classic data structures, due to their multiple dimensions being independent. A possible solution to such problem would be to index the data separately by each of its dimensions. With that said, such solution fails to address either queries that look up multiple dimensions at the same time or queries that are dependent on geometric features of the data, while also having to maintain multiple synchronized copies of the data. In order to efficiently store and optimize algorithms that may use such objects, geometric data structures, such as rtrees, quadtrees, kdtrees and octrees, which subdivide the geometric space, are used. This work implements a library, in the C++ programming language, for the quadtree and kdtree structures, focusing on user customization and documentation in a didactic way, so that it may help other people to implement likewise data structures in a generic and efficient way.*

Resumo. *Objetos geométricos, tais como pontos, retas, polígonos e cubos, não possuem uma ordem intrínseca para estruturas de dados clássicas, devido as suas diversas dimensões serem independentes. Uma possível solução para o armazenamento de dados multidimensionais seria a multi-indexação dos dados por cada uma de suas dimensões. Porém, buscas que, ou utilizem características geométricas dos dados, ou usem mais de uma das diversas dimensões ao mesmo tempo, ainda têm seu desempenho degradada, junto da necessidade de manter diversas cópias atualizadas. Dito isto, como solução para armazenamento e otimização de algoritmos que utilizem estes objetos, estruturas de dados geométricas, tais como rtree, quadtree, kdtree e octree, que particionam o espaço geométrico de busca são utilizadas. Este trabalho visa a implementação, na linguagem C++, de uma biblioteca para as estruturas quadtree e kdtree, com foco na alta personalização dos dados e documentação das técnicas de implementação utilizadas de forma didática, com o intuito de ajudar outras pessoas a implementar estruturas similares de maneira eficiente e genérica.*

1. Introdução

Diferentes problemas computacionais requerem diferentes algoritmos, que processam os dados, e diferentes estruturas de dados, que organizam os dados para uma resolução eficiente. Além disso, diferentes tipos de problemas agrupam-se em diferentes categorias. Uma destas categorias é a de problemas geométricos, que para uma resolução eficiente requer estruturas de dados geométricas.

Problemas geométricos envolvem dados geométricos, tais como pontos, retas e polígonos. Estruturas de dados geométricas, desta forma, se aproveitam de características pertinentes a estes dados para organizar o espaço, diminuindo o espaço de busca, o gasto de memória ou aumentando o desempenho. Técnicas comuns envolvem o agrupamento de dados logicamente próximos, simplificação de regiões em outras menores e repartição do espaço em elementos geométricos específicos. Pode-se citar aqui, por exemplo

1. para o armazenamento de uma imagem binária, usualmente se requer pelo menos um bit por pixel, em que o valor 1 representa a presença de um objeto e o valor 0 representa a ausência. Para redução do custo de memória, pode-se aproveitar a característica de píxeis próximos usualmente terem um valor em comum para definir uma estrutura que simplifica grandes regiões da imagem com o valor 1 para apenas um descritor da região. Regiões com o valor 0 podem ser omitidas, visto que a imagem é binária e apenas 2 valores são possíveis. Desta forma, o consumo pode ser reduzido drasticamente.
2. uma das possíveis otimizações para busca de vizinho mais próximo em bancos de dados é agrupar dados próximos em uma estrutura que preserve logicamente sua proximidade. Desta forma, não é necessário procurar o banco inteiro em consultas de vizinho mais próximo. Isso traz um ótimo ganho de desempenho, visto que não apenas bancos de dados armazenam alta quantidade de dados, assim como também requer acessos à memória secundária, que é ordens de magnitude mais lento que acesso à memória principal.

Para agilizar o desenvolvimento de projetos que necessitam destas estruturas, dados e algoritmos, bibliotecas genéricas são desenvolvidas em diferentes linguagens. Desta forma, cada projeto tem a opção de apenas importar uma destas bibliotecas, sem a necessidade de criar uma própria. Uma destas linguagens é a linguagem C++, que permite abstração de dado e maior controle do hardware ao mesmo tempo. Diferentes bibliotecas têm diferentes objetivos. Algumas bibliotecas, como a [FLA], focam em desempenho e paralelização de algoritmos geométricos de vizinho mais próximo, porém apenas retornando um valor aproximado e necessitando de utilização de herança, com a utilização do padrão de projeto *template* [Gamma et al. 1994]. Outras bibliotecas, como a [boo], focam em usabilidade, extensibilidade e elementos de programação genérica. Este trabalho foca na implementação de um *framework* de estruturas de dados geométricas em C++ com um foco mais didático e de boa usabilidade.

2. Desenvolvimento

Para este trabalho, foi desenvolvido um *framework* na linguagem de programação C++ com quatro diferentes módulos. São estes os módulos de memória, algoritmos, personalização dos dados e estruturas de dados. A visão geral do *framework* pode ser observada conforme a figura 1.

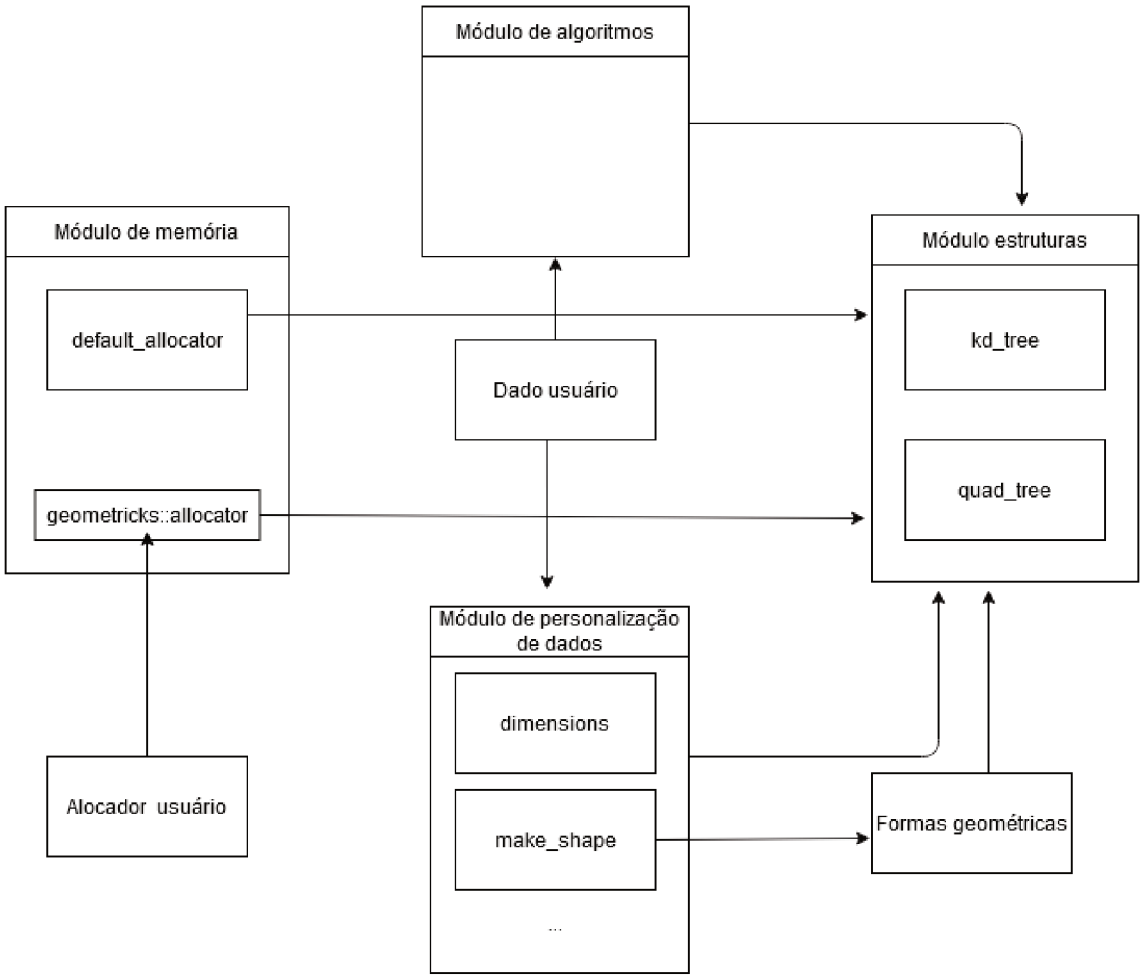

Figura 1. Arquitetura geral do *framework* desenvolvido.

Cada um dos módulos tem como objetivo a não intrusão aos dados do usuário, resolvendo toda a personalização e integração em tempo de compilação utilizando do mecanismo de *template* da linguagem de programação C++. Desta forma, é possível manter as estruturas genéricas o suficiente, mas sem necessidade dos dados do usuário serem acoplados a tipos internos ao *framework*. O módulo principal do *framework* pode ser citado como o módulo de estrutura de dados, com os outros módulos fazendo uma ponte de personalização e sendo utilizados internamente por estes.

As seções a seguir tratam da funcionalidade e desenvolvimento de cada um dos módulos separadamente. A seção 2.1 descreve o funcionamento do módulo de personalização dos dados, brevemente resumindo seu funcionamento. A seção 2.2 trata de diferentes algoritmos e sua implementação como ponto de personalização. A seção 2.3 descreve a estrutura geral de memória, mostrando como a implementação utiliza de técnicas de *type erasure* para criar *virtual tables* em tempo de compilação, emulando técnicas de herança mas eliminando a intrusão inerentes a estas. Por fim, a seção 2.4 brevemente fala sobre as estruturas de dados implementadas, junto com uma pequena descrição da estrutura geral utilizada por estas.

2.1. Módulo de personalização dos dados

O módulo de personalização dos dados disponibiliza de funções para extrações de informações sobre os dados do usuário, se comunicando diretamente com as formas geométricas do *framework* e com o módulo de estrutura de dados. Funções para extrair valores contidos em diferentes dimensões, dado diferentes *tag types* definidos internamente ao *framework* capazes de representar em tempo de compilação uma dimensão específica, junto de especializações de *traits* onde o usuário final informa ao sistema a quantidade de dimensões contidas pelo seu tipo de dado, são disponibilizadas. Também são disponibilizadas neste módulo funções de personalização de alocadores de memória, funções capazes de transformar dados do usuário em formas geométricas internas ao *framework* e vice versa e personalização de funções de distância utilizadas em buscas de *K Nearest Neighbor* (KNN).

De forma geral, este módulo faz a ponte entre os diversos módulos do *framework* com os dados do usuário. Inicialmente, são utilizados protocolos com *Argument Dependent Lookup* (ADL) e técnicas de *Template Meta Programming* (TMP) [Vandevoorde and Josuttis 2003] para detecção de implementações padrão. Caso estas implementações não estejam disponíveis, são dadas opções de disponibilização em `namespaces` específicos. Para a utilização dos outros módulos, então, pelo menos uma destas alternativas deve estar presente, onde são realizadas chamadas diretamente aos pontos de personalização que repassam para a implementação concreta.

2.2. Módulo de algoritmos

Este módulo provê pontos de personalização definidos como lambdas genéricos, utilizando de ADL, para a implementação de algoritmos numéricos e de uso geral, tais como `abs`, `absolute_difference` e `iter_swap`. Esse módulo provê implementações padrão para tipos de dados numéricos, no caso geral tratando dados do usuário como `int`. Caso não sejam definidas as operações de $<$ ou $-$ nestes dados, então, é possível definir em um `namespace` de personalização disponibilizado como parte da biblioteca.

Estes pontos de personalização, em geral, são chamados pelas funções de distância disponibilizadas em outros módulos e internamente no módulo de estruturas de dados. Caso seja necessário mudar o cálculo de distância, retornando um tipo de dado especial na diferença entre dados, como por exemplo em operações em *big ints*, este módulo pode ser utilizado.

2.3. Módulo de memória

O módulo de memória disponibiliza uma interface utilizando de *type erasure* para alocadores de memória personalizados pelo usuário. Um alocador de memória interno do *framework* é apenas uma *view*, contendo um ponteiro ao alocador de memória em si e um ponteiro a uma *virtual table* gerada em tempo de compilação que contém as funções de alocação e desalocação. Estas funções apenas recebem o `void *` do alocador do usuário e dão um `cast` ao tipo concreto, fazendo a chamada correta de alocação de memória disponibilizadas em um protocolo. Devido a este fato, um ponto importante a ser observado é a necessidade do alocador ter tempo de vida útil maior ou igual ao tempo de vida da estrutura de dados que recebe sua referência. Caso essa precondição não seja respeitada, a memória da estrutura é corrompida.

Os alocadores são implementados desta maneira com o intuito de separar o tipo de alocador de memória do tipo de estrutura de dados. No caso do tipo do alocador ser passado por *template*, uma chamada de função que receba uma estrutura ou ficaria presa a um alocador específico, não sendo possível passar a mesma estrutura mas que utilize outro alocador de memória, ou seria necessário a utilização do mecanismo de *template*, com o código inteiro tendo que ser implementado no *header* e causando problemas de *code bloat*.

Também é disponibilizado um alocador padrão de memória. Seu valor inicial, no início de programa, utiliza o `operator new` da linguagem C++ para alocar e desalocar *bytes*. O *framework* então provê funções de atribuir e retribuir o alocador padrão, de forma a dar mais controle ao usuário. Este alocador padrão é passado como argumento para construtores de estruturas que não recebam um alocador específico.

2.4. Módulo de estrutura de dados

São implementadas pelo *framework* as estruturas de dados *kdtree* [Bentley 1975] e *quadtree* [Samet 1984]. Ainda é feito um esboço da estrutura *octree* [Whang et al. 1995], porém, uma implementação concreta desta estrutura não está atualmente disponível. As seções 2.4.1, 2.4.2 e 2.4.3 tratam de técnicas de implementação utilizadas para cada uma destas estruturas.

2.4.1. *KDTree*

A *kdtree* implementada pelo *framework* disponibiliza funções de *nearest neighbor* (NN), KNN, *range query* e construção em *bulk loading*. Devido ao fato de não existirem algoritmos eficientes para balanceamento de *kdtrees*, operações de inserção e remoção de elementos não são suportadas.

Para a implementação da *kdtree*, é utilizado um *array* estático guardando todos os elementos presentes na árvore, onde a raiz se encontra no meio. Cada nó da árvore é representado por um índice I, que índica sua posição no *array* e um tamanho de bloco T, que indica quantos elementos essa subárvore possui. Desta forma, dado um nó $N = (I, T)$, é possível encontrar seus filhos FE e FD conforme as fórmulas 1 e 2,

$$FE \leftarrow \begin{cases} (I - \frac{T}{4} - 1, \frac{T}{2}), & \text{se tamanho bloco é ímpar} \\ (I - \frac{T}{4}, \frac{T}{2}), & \text{caso contrário} \end{cases} \tag{1}$$

$$FD \leftarrow \begin{cases} (I + \frac{T}{4} + 1, \frac{T}{2}), & \text{se tamanho bloco é ímpar} \\ (I + \frac{T}{4}, \frac{T}{2} - 1), & \text{caso contrário} \end{cases} \tag{2}$$

onde FE é o filho da esquerda e FD o filho da direita. Caso o tamanho T seja igual a 1, este nó é considerado um nó raiz, não tendo filho à esquerda nem filho à direita. Uma outra possível observação é a necessidade de armazenar apenas o tamanho da raiz, com todos os outros valores sendo calculados em tempo de execução. Isso torna a árvore extremamente compacta e aumenta seu desempenho, aproveitando melhor as linhas de *cache*. Um exemplo de uma *kdtree* implementada desta forma é observado conforme a figura 2, onde cores mais claras representam nós mais profundos da árvore e as setas

saindo de cada um dos nós representam seus filhos FE e FD calculados de tal maneira. As barras horizontais representam os tamanhos de bloco e, no caso de sua ausência, um tamanho de bloco $T = 1$, representando um nó folha.

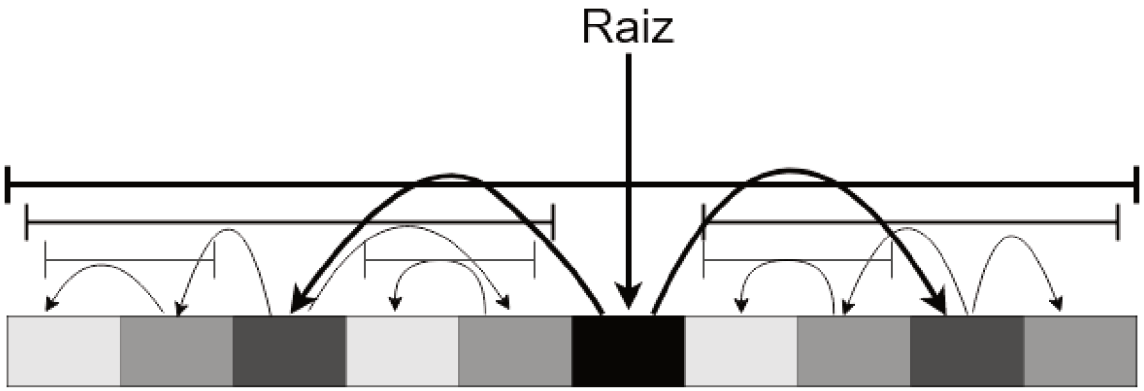

Figura 2. Arquitetura geral do *framework* desenvolvido.

Os métodos recursivos da árvore acessam a dimensão com valores em tempo de compilação. Dado um método recursivo, é passado a dimensão atual D como *template*, sendo o valor da próxima dimensão passado à chamada recursiva $P = (D + 1) \mod K$, onde K é o número de dimensões. O valor inicial de dimensão é sempre 0. Isso otimiza tempo de execução em troca de um executável maior e mais tempo de compilação.

Outra otimização aplicada é de, no método de KNN, um vetor utilizando de *Small Buffer Optimization* (SBO) é prealocado na *stack* para evitar alocação dinâmica de memória.

2.4.2. *Quadtree*

A *quadtree* implementada pelo *framework* utiliza retângulos de dados do tipo `int` para armazenar seus objetos. Caso o usuário queira armazenar tipos de dados diferentes, funções que mapeiem dos dados do usuário para `int` e vice versa são disponibilizadas no módulo de personalização de dados. Por exemplo, se for necessário a utilização de ponto flutuante, é possível transformar este valor de forma que um valor inteiro $I = 1$ seja igual a um valor flutuante $F = .x$, com cada um dos lados do retângulo dados em uma conversão de I para múltiplos do valor F. No caso de objetos não retangulares, a árvore armazena *bounding rects* a estes elementos.

A árvore é implementada em 3 diferentes vetores conforme a figura 3. Os 3 vetores utilizam o mesmo alocador de memória, passado ao construtor da *quadtree*. Para redução de consumo de memória, os *bounding rects* dos nós da árvore são calculados em tempo de execução, tendo apenas o do nó raiz armazenado. Este *boundin grect* é passado como argumento para o construtor e armazenado na árvore. Os retângulos dos subsequentes nós são calculados dividindo o retângulo do nó pai em 4 partes, com o primeiro filho utilizando o retângulo superior esquerdo, o segundo o retângulo superior direito, o terceiro o retângulo inferior esquerdo e o quarto o retângulo inferior direito.

O primeiro vetor armazena os nós da árvore. O elemento 0 do vetor representa

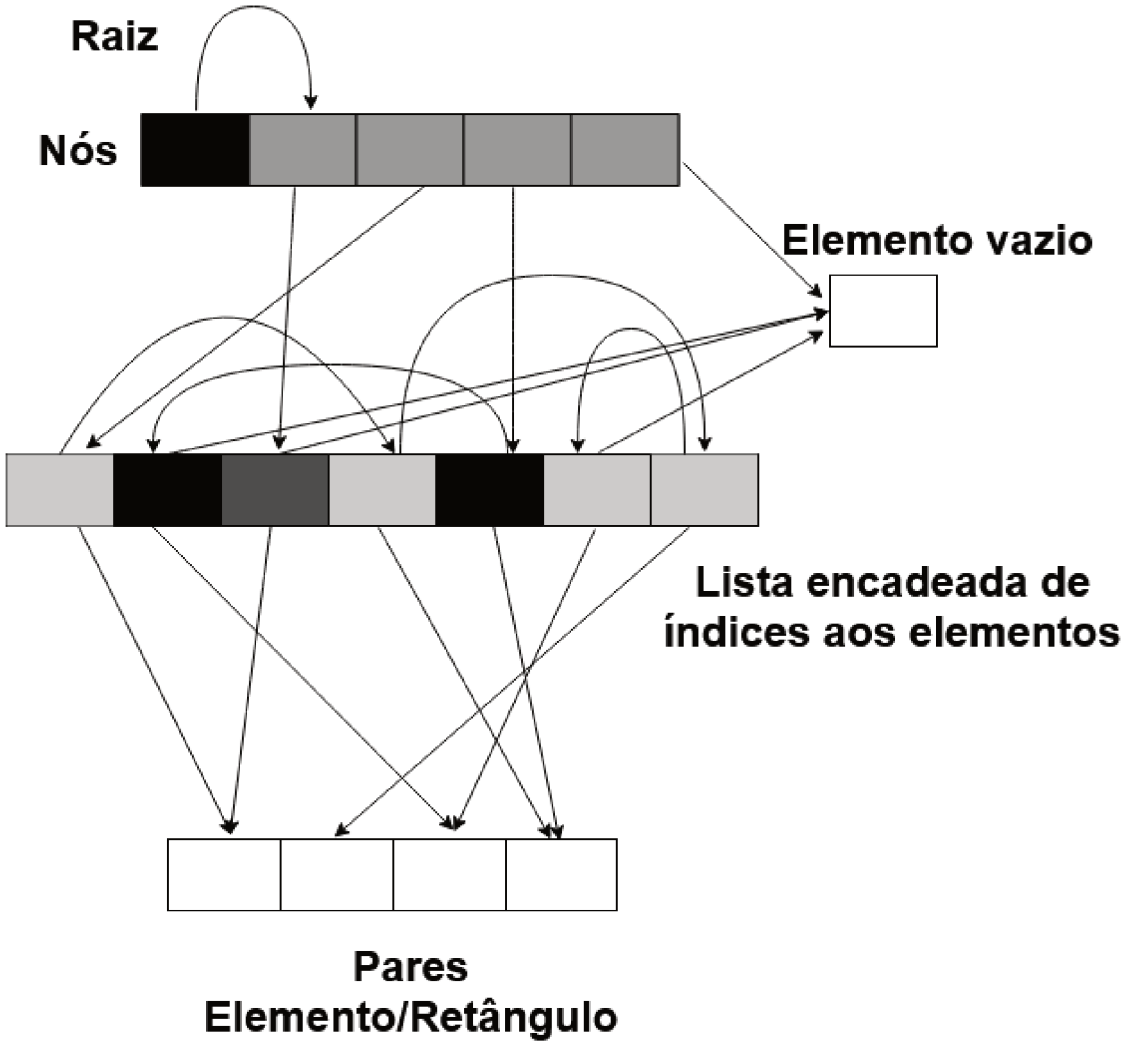

Figura 3. Exemplo de uma estrutura de uma *quadtree* com 4 elementos.

o nó raiz. Cada um destes nós pode armazenar 2 diferentes tipos de valores: um índice para o primeiro nó filho, no caso de um nó intermediário ou um índice para o primeiro elemento, no caso de um nó folha. Cada nó da árvore utiliza apenas um inteiro de tamanho 32 bits. Para diferenciar um nó intermediário de um nó folha, é utilizado o bit mais significativo. Os 31 bits restantes endereçam o elemento contido pelo nó. No caso de um nó intermediário, dado um inteiro de 31 bits I, o *offset* dos 4 filhos no vetor são dados por I, I + 1, I +2 e I + 3. No caso de um nó folha, o valor especial -1 é utilizado para indicar um nó folha vazio. Isso pode ser observado na figura 3. O primeiro elemento representa o nó raiz, um nó intermediário que aponta para o índice 1. Os próximos 3 filhos são encontrados de forma contígua na memória. Os 3 filhos então são nós folhas, apontando para índices no vetor de elementos. O nó no índice 4 contém o valor especial -1, indicando um nó folha vazio e apontando para o elemento vazio.

O segundo vetor é implementado como listas encadeadas de índices para os objetos armazenados em si. A implementação é feita desta maneira pois o mesmo objeto pode ser encontrado em diversos nós folha, caso o seu *bounding rect* intersecte mais de uma folha. Cada elemento armazena 2 índices de 32 bits: o primeiro representa o *offset* para

o próximo elemento da lista encadeada, com o valor especial -1 representando o final da lista e o segundo o *offset* ao vetor de elementos em si. Na figura, cada lista encadeada é representada por um tom diferente. O tamanho máximo de cada lista antes do nó precisar se subdividir em 4 é dado por um parâmetro de *template* passado como *trait* à estrutura, com este tamanho sendo infinito caso o nó não consiga mais se dividir ou um valor de profundidade também passado como argumento de *template* seja atingido. O elemento vazio é apenas simbólico, não sendo necessário armazená-lo.

Por fim, o último vetor contém os objetos armazenados em si, junto com seus *bounding rects*.

2.4.3. *Octree*

A implementação de *octree* é muito parecida com a implementação de *quadtree*, usando as mesmas estruturas descritas na seção 2.4.2. Como diferenças, são apontadas o primeiro vetor da figura 3 tendo 8 filhos ao invés de 4, mas apenas contendo um índice ao primeiro assim como no caso da *quadtree*. Como os objetos armazenados são 3D, os pares do último vetor são de Elemento/Paralelepípedos e as funções auxiliares usadas para a construção da *octree* retornam paralelepípedos no lugar de retângulos. Não há diferenças no resto da implementação significantes a serem mencionadas.

3. Resultados e discussões

Como principal resultado obtido deste trabalho, tem-se o código fonte disponível em `https://github.com/mitthy/TCC`, onde o *framework* sugerido é implementado na linguagem C++.

O *framework* tem como principal objetivo a alta personalização das estruturas. Isso pode ser feito de duas maneiras diferentes: de forma polimórfica dinâmica ou utilizando o mecanismo de *template* da linguagem C++. Com exceção do módulo de memória, o *framework* opta pela utilização dos mecanismos estáticos, devido a forma dinâmica ser intrusiva, com os dados precisando herdar de tipos definidos internamente. Porém, devido ao mecanismo de *template* criar uma cópia da função ou classe para cada definição, isso causa um *tradeoff* no tempo de compilação e tamanho do executável. O módulo de memória utiliza de um polimorfismo *ad-hoc* com técnicas de *type erasure* para separar o tipo de alocador de memória dos tipos de estrutura, podendo tratar estruturas do mesmo tipo de dado mas que recebem alocadores diferentes de forma uniforme às custas de desempenho no tempo de execução.

Enquanto outros trabalhos na área focam mais em indexações com *r-trees*, aproximações de *nearest neighbor* e programações de jogos, o *framework* implementado tem como principal objetivo documentar as diferentes técnicas de implementação utilizadas na criação de estruturas de dados genéricas. Além disso, o *framework* é altamente personalizável e não intrusivo, se assemelhando muito ao Boost.Geometry neste aspecto. Outros trabalhos como FLANN alcançam desempenho melhor, mas são intrusivos aos dados ou apenas retornam dados aproximados com um *trade-off* de sua eficiência.

A estrutura de árvore KD é implementada como um arranjo compacto de forma a salvar memória. Esta representação é possível devido ao fato do número de elemen-

tos da árvore ser constante e a divisão ser sempre feita exatamente na metade. A árvore utiliza vetores alocados na *stack* até um certo número de elemento internamente quando possível para acelerar operações no caso médio, e permite alocadores de memória personalizados. Foram testadas ainda outras alternativas de implementação, utilizando ponteiros para os filhos de cada nó e representando a árvore na forma de uma *heap*. A técnica utilizando ponteiros utilizou mais memória para guardar cada nó, tendo um *overhead* extra do tamanho de 2 ponteiros por elemento, e não foi eficiente com a cache, devido a alocação dos blocos não ser contígua, sendo então descartada. A técnica utilizando uma representação de *heap* gastou muito mais memória que precisava. Outras considerações tomadas à árvore dizem respeito ao armazenamento de chaves e valores, possibilitando o uso desta como um mapa. Pelo levantamento, esta estrutura é utilizada para dados de não muitas dimensões, em aplicações onde não acontecem muitas inserções dinâmicas e sim bastantes operações de busca.

Para a implementação de *quadtree* e *octree*, foi escolhido apenas manter índices aos elementos nos nós e guardá-los em uma lista de vetor encadeada. Isso se dá devido ao fato de que guardar um vetor de elementos nos nós é muito mais custoso em memória, precisando guardar metadados dos elementos em cada um dos nós internos. Guardando apenas um índice, é possível diminuir o uso de memória de cada um dos nós para apenas 32 bits. Outra consideração que poderia aumentar um pouco o uso de memória mas acelerar os algoritmos seria armazenar informações tais como o número de elementos de cada nó, não precisando calcular dinamicamente este valor. A árvore ainda permite a inserção de diferentes elementos no mesmo ponto, permitindo mais flexibilização, mas com perda de desempenho e necessitando cuidar de casos de infinitas subdivisões. Para isso, são usados valores externos indicando a profundidade máxima e o número máximo de elementos por nó. Cada nó guarda apenas um índice ao seu primeiro filho, devido ao fato da divisão de espaço sempre ocorrer de 4 em 4 filhos, é possível achar os próximos filhos dinamicamente sem muitos custos e diminuir o gasto de memória. Essa representação torna possível identificar se um nó é folha ou não utilizando apenas 1 bit, enquanto os outros 31 bits são utilizados para indexar. Outra consideração feita foi de utilizar a árvore como um mapeamento de retângulos para objetos. Como a árvore *octree* é muito parecida com a *quadtree*, as técnicas e considerações de implementação são as mesmas. Essas estruturas permitem uma inserção e remoção eficiente, com o levantamento mostrando que são utilizadas em aplicações que necessitam de criação, atualização e busca de diversos objetos, como jogos que têm milhares de partículas na tela e precisam de otimizações para verificação de colisão de objetos, assim como para representar regiões de espaço que contenham uniformemente o mesmo valor. Essa última aplicação não é suportada pela implementação realizada no *framework*.

A interface utiliza de objetos ponto de personalização para as operações personalizáveis, *views* para alocadores de memórias e funções membro dentro das classes das estruturas. Uma possível consideração é extrair algumas funções de busca como funções livre, criando um *overloadset* para as estruturas em geral. Os pontos de personalização então verificam chamadas de acordo com o protocolo e não sofrem *override* em si, evitando a necessidade do uso de `using` como encontrado em `swap`. A interface das estruturas em si possuem a parte estática de *template* e a parte das funções. Enquanto *templates* são passados como argumento para o tipo de dado em si, permitindo a personalização de operações básicas em tempo de compilação, as funções aceitam quando possível objetos

callable para flexibilizar as operações.

4. Conclusão

O desenvolvimento deste trabalho disponibilizou uma biblioteca *open source* para estruturas de dados geométricas na linguagem C++. Um levantamento inicial de quais estruturas e algoritmos a serem implementados levou a decisão de implementação de*KD trees*, *quadtrees*, *octrees* e *rtrees*, devido a essas estruturas serem mais clássicas na literatura e mais simples de entender a implementação, o que facilita no objetivo de documentar de forma didática o documento, focando mais nas técnicas utilizadas e não em detalhes das estruturas.

Um estudo de cada uma destas estruturas foi feito, com diferentes técnicas de implementação revisadas e ponderadas para cada uma delas, assim como uma revisão em outros *frameworks* similares para este tipo de estrutura. Depois de tomadas as decisões de implementações, paralelamente com a criação deste documento, as estruturas *kd tree* e *quadtree* foram implementadas na linguagem C++, assim como um *framework* de suporte para personalização de dados e gerenciamento de memória, possibilitando ao usuário definir diferentes padrões de acesso. Outras estruturas de baixo nível são implementadas internamente para otimizações de algoritmos, operações e para fazer uma ponte entre diferentes componentes do *framework*. Um conjunto de testes utilizando Google Test ainda é implementado para fins de testes de corretude e usabilidade de cada uma das implementações.

Para trabalhos futuros, tem-se a implementação das estruturas *octree*, *rtree* e *grid*, assim como a possibilidade de utilizar as estruturas atualmente implementadas como um mapa, melhorando a usabilidade do *framework*. Outras sugestões de trabalhos futuros incluem a inclusão de implementações de outras métricas de distância, *benchmarks* com outras implementações, implementação de tipos concretos de alocadores de memória, otimizações de baixo nível nas estruturas existentes, inclusão de novas formas geométricas e otimização de parâmetros estáticos para estruturas de baixo e alto nível definidos no código.

Referências

Boost.geometry.

Flann - fast library for approximate nearest neighbors.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns*. Addison-Wesley.

Samet, H. (1984). The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260.

Vandevoorde, D. and Josuttis, N. M. (2003). *C++ Templates: The Complete Guide*. Addison-Wesley Professional.

Whang, K.-Y., Song, J.-W., Chang, J.-W., Kim, J.-Y., Cho, W.-S., Park, C.-M., and Song, I.-Y. (1995). Octree-r: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349.

APÊNDICE B – Código Fonte

Código B.1 – geometricks/algorithm/absolute_difference.hpp

```
1  #ifndef GEOMETRICKS_ALGORITHM_ABSOLUTE_DIFFERENCE_HPP
2  #define GEOMETRICKS_ALGORITHM_ABSOLUTE_DIFFERENCE_HPP
3
4  //C stdlib includes
5  #include <stdint.h>
6  #include <cmath>
7
8  //C++ stdlib includes
9  #include <algorithm>
10 #include <type_traits>
11
12 //Project includes
13 #include "geometricks/meta/utils.hpp"
14
15 /**
16 * @file
17 * @brief Provides abs and absolute difference functions along
       with customization options used within the library.
18 */
19
20 namespace geometricks {
21
22   namespace algorithm {
23
24     namespace abs_customization {
25
26       /** @brief
27       Struct providing meta function for calling abs on a type
           supplied by the user.
28       @tparam T Type to generate custom abs function.
29       @details To use this function, specialize it for your type
            and provide a static method named _ that takes a
           variable of your type.
30       @warning A definition of this struct for a type shadows
```

```
        all other possible invocations for abs on that type.
31      @see abs_t
32
33      Example:
34      @code{.cpp}
35      namespace dummy {
36        struct dummy{
37          int m_value;
38        };
39      }
40      namespace geometricks {
41        namespace algorithm {
42          namespace abs_customization {
43            template<>
44            struct abs<dummy::dummy> {
45              static int _( const dummy& dum ) {
46                return std::abs( dum.m_value );
47              }
48            };
49          }
50        }
51      }
52      @endcode
53      */
54      template< typename T >
55      struct abs;
56
57    }
58
59    /**
60    * @cond EXCLUDE_DOXYGEN
61    *
62    * Internal not to be documented
63    */
64    namespace __detail__ {
65
66      template< typename T >
67      using is_unsigned = std::enable_if_t<std::is_unsigned_v<T
          >, bool>;
```

```
68
69          template< typename T >
70          using is_signed = std::enable_if_t<std::is_signed_v<T>,
                bool>;
71
72          using std::abs;
73
74          template< typename T >
75          constexpr auto
76          __abs__( const T& value, meta::priority_tag<0> ) ->
                decltype( value < 0 ? -value : value ) {
77            return value < 0 ? -value : value;
78          }
79
80          template< typename T, is_unsigned<T> = true >
81          constexpr size_t
82          __abs__( const T& value, meta::priority_tag<1> ) {
83            return value;
84          }
85
86          template< typename T >
87          constexpr auto
88          __abs__( const T& value, meta::priority_tag<2> ) ->
                decltype( static_cast<size_t>( abs( value ) ) ) {
89            return static_cast<size_t>( abs( value ) );
90          }
91
92          template< typename T >
93          constexpr auto
94          __abs__( const T& value, meta::priority_tag<3> ) ->
                decltype( abs_customization::abs<T>::_( value ) ) {
95            return abs_customization::abs<T>::_( value );
96          }
97
98        }
99        /**
100       * @endcond
101       */
102
```

```
103     /**
104     @brief Functor that computes the abs of a number
105
106     Functor for abs that provides automatic ADL for the user.
107     */
108     struct abs_t {
109
110       /**
111       @brief Calculates the abs of a number.
112       @param value Number to calculate the absolute value.
113       @return The absolute value.
114       @details The abs specification goes as follows:
115         -# If a specialization of abs_customization::abs is
              provided for that type, the specialization _function
               is called.
116         -# If an abs function is found via ADL for that type,
              the abs function is called.
117         -# If the type is unsigned, it is returned as is.
118         -# If a < operator for the type exists the < operator
              compares to an int and a - unary operator for that
              type exists, it returns @code{.cpp} value < 0 ? -
              value : value; @endcode
119
120       * If all fails, the call is ill-formed.
121       */
122       template< typename T >
123       constexpr auto
124       operator()( const T& value ) const -> decltype( __detail__
            ::__abs__( value, meta::priority_tag<3>{} ) ) {
125         return __detail__::__abs__( value, meta::priority_tag
              <3>{} );
126       }
127
128     };
129
130     /**
131     @brief Function object for @relatealso abs_t
132     */
133     constexpr auto
```

```
134     abs = abs_t{};
135
136     namespace absolute_difference_customization {
137
138       /**@brief
139       Struct providing meta function for calling the absolute
            difference on a type supplied by the user.
140       @tparam T Type to generate custom abs function.
141       @details To use this function, specialize it for your type
             and provide a static method named _ that takes 2
            variables of your type.
142       @warning A definition of this struct for a type shadows
            all other possible invocations for absolute difference
            on that type.
143       @see absolute_difference_t
144
145       Example:
146       @code{.cpp}
147       namespace dummy {
148         struct dummy{
149           int m_value;
150         };
151       }
152       namespace geometricks {
153         namespace algorithm {
154           namespace absolute_difference_customization {
155             template<>
156             struct absolute_difference<dummy::dummy> {
157               static dummy _( const dummy& lhs, const dummy& rhs
                      ) {
158                 return std::abs( lhs.m_value - rhs.m_value );
159               }
160             };
161           }
162         }
163       }
164       @endcode
165       */
166       template< typename T >
```

```
167         struct absolute_difference;
168
169       }
170
171       /**
172       * @cond EXCLUDE_DOXYGEN
173       *
174       * Internal not to be documented
175       */
176       namespace __detail__ {
177
178         template< typename T >
179         using is_signed = std::enable_if_t<std::is_signed_v<T>,
              bool>;
180
181         template< typename T >
182         using is_unsigned = std::enable_if_t<std::is_unsigned_v<T
              >, bool>;
183
184         template< typename T >
185         using is_string = std::enable_if_t<std::is_same_v<T, std::
              string>, bool>;
186
187         template< typename T >
188         constexpr size_t
189         __absolute_difference__( T first, T second, meta::
              priority_tag<0> ) {
190           return algorithm::abs( first - second );
191         }
192
193         template< typename T, is_unsigned<T> = true >
194         constexpr size_t
195         __absolute_difference__( T first, T second, meta::
              priority_tag<1> ) {
196           return first > second ? first - second : second - first;
197         }
198
199         template< typename T, is_string<T> = true >
200         constexpr size_t
```

```
201         __absolute_difference__( T first, T second, meta::
                priority_tag<2> ) {
202           return 0; //TODO
203         }
204
205         template< typename T >
206         constexpr auto
207         __absolute_difference__( T first, T second, meta::
                priority_tag<3> ) -> decltype(
                absolute_difference_customization::absolute_difference<
                T>::_( first, second ) ) {
208           return absolute_difference_customization::
                    absolute_difference<T>::_( first, second );
209         }
210
211       } //namespace __detail__
212       /**
213       * @endcond
214       */
215
216       /**
217       @brief Functor that computes the absolute difference of 2
              number
218
219       Functor for absolute difference that provides automatic ADL
              for the user.
220       */
221       struct absolute_difference_t {
222
223         /**
224         * @brief Computes the absolute difference of 2 numbers.
225         * @tparam T Type of the first argument. Should have same
              decay type as U.
226         * @tparam U Type of the second argument. Should have save
              decay type as T.
227         * @param first First of the 2 numbers to calculate
              absolute difference.
228         * @param second Second of the 2 numbers to calculate
              absolute difference.
```

```
229         * @details The absolute difference of 2 numbers is the
                positive difference between those 2 numbers.
230         * That is, absolute_difference( a, b ) >= 0.
231         * \n To get this effect, some techniques can be employed
                in order:
232         * -# If a specialization of
                absolute_difference_customization::absolute_difference
                is provided for that type, the specialization _
                function is called.
233         * -# If the type is a string, we get the edit distance.
234         * -# If the type is unsigned, we get the greater of the 2
                numbers and subtract the lesses number.
235         * -# If it is possible to call abs on the number and there
                 is an operator -, we call abs on the difference.
236         *
237         * If all fails, the call is ill-formed.
238         * @todo Implement string edit distance.
239         */
240        template< typename T, typename U >
241        constexpr auto operator()( T&& first, U&& second ) const
              ->
242        decltype( __detail__::__absolute_difference__( std::
              forward<T>( first ), std::forward<T>( second ), meta::
              priority_tag<3>{} ) ) {
243          static_assert( std::is_same_v<std::decay_t<T>, std::
                decay_t<U>> );
244          return __detail__::__absolute_difference__( std::forward
                <T>( first ), std::forward<T>( second ), meta::
                priority_tag<3>{} );
245        }
246
247    };
248
249    /**
250    @brief Function object for @relatealso absolute_difference_t
251    */
252    constexpr auto
253    absolute_difference = absolute_difference_t{};
254
```

```
255     } //namespace algorithm
256
257 } //namespace geometricks
258
259 #endif //GEOMETRICKS_ALGORITHM_ABSOLUTE_DIFFERENCE_HPP
```

Código B.2 – geometricks/algorithm/iter_swap.hpp

```
1  #ifndef GEOMETRICKS_ALGORITHM_ITER_SWAP_HPP
2  #define GEOMETRICKS_ALGORITHM_ITER_SWAP_HPP
3
4  //Project includes
5  #include "geometricks/meta/utils.hpp"
6  #include <type_traits>
7
8  /**
9  * @file
10 * @brief Provides definitions for iter_swap function.
11 */
12
13 namespace geometricks {
14
15   namespace algorithm {
16
17     namespace iter_swap_customization {
18
19       /**@brief
20       Struct providing meta function for calling iter_swap on a
             type supplied by the user.
21       @tparam T Type to generate custom iter_swap function.
22       @details To use this function, specialize it for your type
              and provide a static method named _ that takes a
             variable of your type.
23       @warning A definition of this struct for a type shadows
             all other possible invocations for iter_swap on that
             type.
24       @see iter_swap_t
25
26       Example:
27       @code{.cpp}
28       namespace dummy {
29         struct dummy_it{
30           int* m_value;
31         };
32       }
33       namespace geometricks {
```

```
34          namespace algorithm {
35            namespace iter_swap_customization {
36              template<>
37              struct iter_swap<dummy::dummy> {
38                static constexpr void _( dummy left, dummy right )
                      {
39                  std::swap( *left.m_value, *right.m_value );
40                }
41              };
42            }
43          }
44        }
45        @endcode
46        */
47        template< typename T >
48        struct iter_swap;
49
50      }
51
52      /**
53      * @cond EXCLUDE_DOXYGEN
54      *
55      * Internal not to be documented
56      */
57      namespace __detail__ {
58
59        using std::iter_swap;
60
61        using std::swap;
62
63        template< typename Iter1, typename Iter2 >
64        constexpr auto
65        __iter_swap__( Iter1 left, Iter2 right, meta::priority_tag
              <0> ) -> std::void_t<decltype( swap( *left, *right ) )>
               {
66          swap( *left, *right );
67        }
68
69        template< typename Iter1, typename Iter2 >
```

```
70        constexpr auto
71        __iter_swap__( Iter1 left, Iter2 right, meta::priority_tag
              <1> ) -> std::void_t<decltype( iter_swap( left, right )
               )> {
72          iter_swap( left, right );
73        }
74
75        template< typename Iter1, typename Iter2 >
76        constexpr auto
77        __iter_swap__( Iter1 left, Iter2 right, meta::priority_tag
              <2> ) -> std::void_t<decltype( iter_swap_customization
               ::iter_swap<Iter2>::_( left, right ) )> {
78          iter_swap_customization::iter_swap<Iter2>::_( left,
                right );
79        }
80
81        template< typename Iter1, typename Iter2 >
82        constexpr auto
83        __iter_swap__( Iter1 left, Iter2 right, meta::priority_tag
              <3> ) -> std::void_t<decltype( iter_swap_customization
               ::iter_swap<Iter1>::_( left, right ) )> {
84          iter_swap_customization::iter_swap<Iter1>::_( left,
                right );
85        }
86
87      }
88      /**
89      * @endcond
90      *
91      */
92
93      /**
94      @brief Functor that swaps 2 iterators.
95      *
96      * Functor for iter_swap that provides automatic ADL for the
            user.
97      */
98      struct iter_swap_t {
99
```

```
100         /**
101         * @brief Swaps the value held by the iterators.
102         * @tparam Iter1 The type of the left iterator.
103         * @tparam Iter2 The type of the right iterator.
104         * @param left First of the 2 iterators.
105         * @param right Last of the 2 iterators.
106         * @details To swap the values, it tries different things
                in order:
107         * -# It checks for a template specialization of the
                iter_swap struct for Iter1.
108         * -# It checks for a template specialization of the
                iter_swap struct for Iter2.
109         * -# It tries to call iter swap via ADL.
110         * -# It tries calling swap via ADL accessing the
                underlying elements with the * unary operator.
111         *
112         * If all fails, the call is ill-formed.
113         */
114         template< typename Iter1, typename Iter2 >
115         constexpr auto operator()( Iter1 left, Iter2 right ) const
              -> std::void_t<decltype(__detail__::__iter_swap__(
              left, right, meta::priority_tag<2>{} ) )> {
116           __detail__::__iter_swap__( left, right, meta::
                priority_tag<2>{} );
117         }
118
119       };
120
121       /**
122       @brief Function object for @relatealso iter_swap_t
123       */
124       constexpr auto iter_swap = iter_swap_t{};
125
126     }
127
128 }
129
130 #endif //GEOMETRICKS_ALGORITHM_ITER_SWAP_HPP
```

Código B.3 – geometricks/algorithm/log.hpp

```
1  #ifndef GEOMETRICKS_ALGORITHM_LOG_H
2  #define GEOMETRICKS_ALGORITHM_LOG_H
3
4  /**
5  * @file
6  * @brief Provides constexpr log2 function to be used in
       templates.
7  */
8
9  //C stdlib includes
10 #include <stdint.h>
11
12 namespace geometricks {
13
14   namespace algorithm {
15
16     namespace utils {
17
18       /**
19       * @brief Computes the log base 2 of a number in a
           constexpr way.
20       * @param input The input number.
21       * @return log base 2 of the input rounded down to an
           integer.
22       */
23       constexpr int8_t
24       log2( uint64_t input ) {
25         if( !input ) {
26           return -1;
27         }
28         int8_t result = 0;
29         while( input >>= 1 ) {
30           ++result;
31         }
32         return result;
33       }
34
35     } //namespace utils
```

```
36
37    }  //namespace data_structure
38
39 }  //namespace geometricks
40
41
42
43 #endif //GEOMETRICKS_ALGORITHM_LOG_H
```

Código B.4 – geometricks/algorithm/mean.hpp

```
1  #ifndef GEOMETRICKS_ALGORITHM_MEAN_HPP
2  #define GEOMETRICKS_ALGORITHM_MEAN_HPP
3
4  //C++ stdlib includes
5  #include <numeric>
6  #include <algorithm>
7  #include <iterator>
8
9  /**
10 * @file
11 * @brief Provides mean statistic function
12 *
13 * @details some geometric data structures can be optimized if we
       know the mean of the input data in advance.
14 * This file provides the mean functor along with helper traits.
15 */
16
17 namespace geometricks {
18
19   namespace algorithm {
20
21     /**
22     * @brief Type trait to get the number 0
23     * @tparam T Numeric like type.
24     * @details This trait is used by the mean function to init
          its internal value before additions.
25     *
26     * Example:
27     * @code{.cpp}
28     * struct BigInt {
29         constexpr BigInt( const char * value );
30         ...
31       };
32       ...
33     * template< >
34     * struct zero_traits<BigInt> {
35         static constexpr BigInt zero() {
36           return BigInt( "0" );
```

```
37          }
38        };
39        @endcode
40      */
41      template< typename T >
42      struct zero_traits {
43        static constexpr T zero() {
44          return 0;
45        }
46      };
47
48      /**
49      @brief Functor that computes the mean of a range of numbers.
50      */
51      struct mean_t {
52
53        /**
54        * @brief Computes the mean of a range of number.
55        * @param first The begin of a range of elements.
56        * @param last Another iterator pointing to the end of the
              range or a sentinel value that eventually compares
              false with first.
57        * @pre If first and last are both iterators, first < last.
              Else, first != last eventually compares true.
58        * @details Computes the mean of the elements in the range
              [first, last).
59        */
60        template< typename Iterator, typename Sentinel >
61        auto operator()( Iterator first, Sentinel last ) const {
62          using return_type = typename std::iterator_traits<
                Iterator>::value_type;
63          auto size = std::distance( first, last );
64          auto init = zero_traits<return_type>::zero();
65          for( int i = 0; i < size; ++i ) {
66            init += *first;
67            ++first;
68          }
69          return init / size;
70        }
```

```
71
72      };
73
74      /**
75      @brief Function object for @relatealso mean_t
76      */
77      constexpr auto mean = mean_t{};
78
79  }  //namespace algorithm
80
81 }  //namespace geometricks
82
83 #endif //GEOMETRICKS_ALGORITHM_MEAN_HPP
```

Código B.5 – geometricks/algorithm/partition.hpp

```
1  #ifndef GEOMETRICKS_ALGORITHM_PARTITION_HPP
2  #define GEOMETRICKS_ALGORITHM_PARTITION_HPP
3
4  //C++ stdlib includes
5  #include <algorithm>
6  #include "iter_swap.hpp"
7
8  //Reimplementation of std partition where std iter_swap is a
       customisation point to work with custom iterators provided by
        the library.
9  /**
10 * @file
11 * Implements partition algorithm with a custom iter_swap
       predicate.
12 */
13
14 namespace geometricks {
15
16   namespace algorithm {
17
18     /**
19     * @brief Partitions a range
20     * @param first First element of the range.
21     * @param last Last element or sentinel value.
22     * @param func Compare function.
23     * @pre If Sentinel is an iterator, first < last. Else,
           eventually first != last compares false.
24     * @details Partitions a range such that every element that
           compares false for func is to the left of everyone who
           compares true.
25     */
26     template< typename InputIterator, typename Sentinel,
           typename Function >
27     InputIterator partition( InputIterator first, Sentinel last,
          Function func ) {
28       while( first != last && func( *first ) ) {
29         ++first;
30       }
```

```
31        if( first == last ) return first;
32        for( auto next = std::next( first ); next != last; ++next
              ) {
33          if( func( *next ) ) {
34            geometricks::algorithm::iter_swap( first, next );
35            ++first;
36          }
37        }
38        return first;
39      }
40
41    }
42
43  }
44
45  #endif //GEOMETRICKS_ALGORITHM_PARTITION_HPP
```

Código B.6 – geometricks/data_structure/internal/small_vector.hpp

```
1  #ifndef GEOMETRICKS_DATA_STRUCTURE_PRIMITIVES_SMALL_VECTOR_HPP
2  #define GEOMETRICKS_DATA_STRUCTURE_PRIMITIVES_SMALL_VECTOR_HPP
3
4  #include <stdint.h>
5  #include <type_traits>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <memory>
9  #include "utils.hpp"
10 #include "geometricks/memory/allocator.hpp"
11
12 namespace geometricks {
13
14   template< typename T >
15   struct vector {
16
17     using iterator = T*;
18
19     using const_iterator = const T*;
20
21     using value_type = T;
22
23     using reference = T&;
24
25     using const_reference = const T&;
26
27     using size_type = int;
28
29     vector() = delete;
30
31     vector&
32     operator=( const vector& rhs ) {
33       if( &rhs != this ) {
34         if( size() >= rhs.size() ) {
35           auto it = std::copy( rhs.begin(), rhs.end(), begin() )
                ;
36           __destroy__( it, end() );
37         }
```

```
38          else {
39            if( m_capacity < rhs.size() ) {
40              __destroy__( begin(), end() );
41              m_size = 0;
42              __grow__( rhs.size() );
43              std::uninitialized_copy( rhs.begin(), rhs.end(),
                    begin() );
44            }
45            else {
46              std::copy( rhs.begin(), rhs.begin() + m_size, begin
                    () );
47              std::uninitialized_copy( rhs.begin() + m_size, rhs.
                    end(), begin() + m_size );
48            }
49          }
50          m_size = rhs.size();
51        }
52        return *this;
53      }
54
55      vector&
56      operator=( vector&& rhs ) {
57        if( &rhs != this ) {
58          if( !rhs.__is_stack__() ) {
59            __destroy__( begin(), end() );
60            if( !__is_stack__() ) {
61              m_allocator.deallocate( m_data );
62            }
63            m_data = rhs.m_data;
64            m_size = rhs.m_size;
65            m_capacity = rhs.m_capacity;
66            m_allocator = rhs.m_allocator;
67            rhs.__reset_stack__();
68            rhs.set_size( 0 );
69          }
70          else {
71            if( size() >= rhs.size() ) {
72              auto it = std::move( rhs.begin(), rhs.end(), begin()
                      );
```

```
73              __destroy__( it , end() );
74            }
75            else {
76              if( m_capacity < rhs.size() ) {
77                __destroy__( begin(), end() );
78                m_size = 0;
79                __grow__( rhs.size() );
80                std::uninitialized_copy( std::make_move_iterator(
                      rhs.begin() ), std::make_move_iterator( rhs.end
                      () ), begin() );
81              }
82              else {
83                std::move( rhs.begin(), rhs.begin() + m_size,
                      begin() );
84                std::uninitialized_copy( std::make_move_iterator(
                      rhs.begin() + m_size ), std::make_move_iterator
                      ( rhs.end() ), begin() );
85              }
86            }
87            rhs.clear();
88            m_size = rhs.size();
89          }
90        }
91        return *this;
92      }
93
94      size_type
95      push_back( const T& element ) {
96        if( m_size == ( int )m_capacity ) {
97          __grow__( m_capacity + 1 );
98        }
99        if constexpr( std::is_trivially_copyable_v<T> ) {
100         memcpy( ( void* )&m_data[ m_size ], ( void* )&element,
                    sizeof(T) );
101       }
102       else {
103         new ( &m_data[ m_size ] ) T( element );
104       }
105       ++m_size;
```

```
106       return m_size - 1;
107     }
108
109     void
110     pop_back() {
111       m_data[ --m_size ].~T();
112     }
113
114     ~vector() {
115       __destroy__( begin(), end() );
116       if( !__is_stack__() ) {
117         m_allocator.deallocate( m_data );
118       }
119     }
120
121     iterator
122     erase( const_iterator pos ) {
123       pos->~T();
124       iterator old = ( iterator )pos++;
125       std::move( pos, static_cast<const_iterator>( end() ), old
              );
126       --m_size;
127       return old;
128     }
129
130     iterator
131     erase( const_iterator begin, const_iterator last ) {
132       iterator old = ( iterator )begin;
133       int distance = last - begin;
134       __destroy__( begin, last );
135       std::move( last, static_cast<const_iterator>( end() ), old
              );
136       m_size -= distance;
137       return old;
138     }
139
140     iterator
141     begin() {
142       return m_data;
```

```
143     }
144
145     iterator
146     end() {
147       return m_data + m_size;
148     }
149
150     const_iterator
151     begin() const {
152       return m_data;
153     }
154
155     const_iterator
156     end() const {
157       return m_data + m_size;
158     }
159
160     iterator
161     data() {
162       return m_data;
163     }
164
165     const_iterator
166     data() const {
167       return m_data;
168     }
169
170     T&
171     operator[]( int index ) {
172       return m_data[ index ];
173     }
174
175     const T&
176     operator[]( int index ) const {
177       return m_data[ index ];
178     }
179
180     void
181     set_size( int sz ) {
```

```
182       //UNSAFE!! Only call this function if you're sure about
              the new size!!!!!!!
183       //Should be used for [] or data() initialization!
184       m_size = sz;
185     }
186
187     int
188     size() const {
189       return m_size;
190     }
191
192     void
193     reserve( unsigned request ) {
194       if( m_capacity < request ) {
195         __grow__( request );
196       }
197     }
198
199     T&
200     front() {
201       return *m_data;
202     }
203
204     const T&
205     front() const {
206       return *m_data;
207     }
208
209     T&
210     back() {
211       return m_data[ m_size - 1 ];
212     }
213
214     const T&
215     back() const {
216       return m_data[ m_size - 1 ];
217     }
218
219     bool
```

```
220     empty() const {
221       return !m_size;
222     }
223
224     void
225     clear() {
226       __destroy__( begin(), end() );
227       m_size = 0;
228     }
229
230   protected:
231
232     vector( int capacity, geometricks::allocator alloc ): m_data
          ( __get_stack_address__() ),
233                                                           m_size
                                                              ( 0
                                                               ),
234                                                           m_capacity
                                                              (
                                                              capacity
                                                               ),
235                                                           m_allocator
                                                              (
                                                              alloc
                                                               )
                                                              {}
236
237   private:
238
239     struct __stack_ptr_calculator__ {
240       T* _1;
241       int _2;
242       unsigned _3;
243       geometricks::allocator _4;
244       union small_element_t {
245
246         struct dummy {} _;
247
248         T data;
```

```
249
250       };
251       small_element_t _stack[1];
252     };
253
254     T* m_data;
255
256     int m_size;
257
258     unsigned m_capacity;
259
260     geometricks::allocator m_allocator;
261
262     T*
263     __get_stack_address__() const {
264       return static_cast<T*>( static_cast<void*>( &( (
              __stack_ptr_calculator__* ) this )->_stack ) );
265     }
266
267     void
268     __reset_stack__() {
269       m_data = __get_stack_address__();
270     }
271
272     void
273     __destroy__( const T* begin, const T* end ) {
274       if constexpr( !std::is_trivially_destructible_v<T> ) {
275         while( begin != end ) {
276           begin->~T();
277           ++begin;
278         }
279       }
280       else {
281         ( void )begin;
282         ( void )end;
283       }
284     }
285
286     bool
```

```
287     __is_stack__() const {
288       return m_data == __get_stack_address__();
289     }
290
291     void
292     __grow__( unsigned request ) {
293       unsigned new_capacity = __detail__::next_power_of_2(
              request );
294       new_capacity = std::max( 1u, new_capacity );
295       if constexpr( std::is_trivially_copyable_v<T> ) {
296         if( __is_stack__() ) {
297           T* heap_mem = static_cast<T*>( m_allocator.allocate(
                  sizeof( T ) * new_capacity ) );
298           memcpy( heap_mem, m_data, m_size * sizeof( T ) );
299           //No need to free the buffer since it was on the stack
                  .
300           //Also, safe to leave memory as is. Trivially copyable
                    types don't need destructor calls.
301           m_data = heap_mem;
302           m_capacity = new_capacity;
303         }
304         else {
305           T* heap_mem = static_cast<T*>( m_allocator.allocate(
                  sizeof( T ) * new_capacity ) );
306           memcpy( heap_mem, m_data, m_size * sizeof( T ) );
307           m_allocator.deallocate( m_data );
308           m_data = heap_mem;
309           m_capacity = new_capacity;
310         }
311       }
312       else {
313         T* new_mem = static_cast<T*>( m_allocator.allocate(
                sizeof( T ) * new_capacity ) );
314         std::uninitialized_copy( std::make_move_iterator( m_data
                ), std::make_move_iterator( m_data + m_size ),
              new_mem );
315         __destroy__( begin(), end() );
316         if( !__is_stack__() ) {
317           m_allocator.deallocate( m_data );
```

```
318           }
319           m_data = new_mem;
320           m_capacity = new_capacity;
321         }
322       }
323
324   };
325
326   template< typename T, int SBOSize >
327   class small_vector : public vector<T> {
328
329     static_assert( SBOSize >= 0 );
330
331     union small_element_t {
332
333       struct dummy {} _;
334
335       T data;
336
337       small_element_t() {
338         _ = dummy{};
339       }
340
341     };
342
343     struct small_t {
344
345       small_element_t m_data[ SBOSize ];
346
347     };
348
349     small_t m_stack;
350
351   public:
352
353     small_vector( geometricks::allocator alloc = geometricks::
          allocator{} ): vector<T>::vector( SBOSize, alloc ) {
354     }
355
```

```
356     small_vector( const vector<T>& other, geometricks::allocator
            alloc = geometricks::allocator{} ): vector<T>::vector(
           SBOSize, alloc ) {
357       vector<T>::operator=( other );
358     }
359
360     small_vector( vector<T>&& other, geometricks::allocator
           alloc = geometricks::allocator{} ): vector<T>::vector(
           SBOSize, alloc ) {
361       vector<T>::operator=( std::move( other ) );
362     }
363
364     small_vector& operator=( const geometricks::vector<T>& other
            ) {
365       vector<T>::operator=( other );
366       return *this;
367     }
368
369     small_vector& operator=( geometricks::vector<T>&& other ) {
370       vector<T>::operator=( std::move( other ) );
371       return *this;
372     }
373
374   };
375
376 }
377
378 #endif //GEOMETRICKS_DATA_STRUCTURE_PRIMITIVES_SMALL_VECTOR_HPP
```

Código B.7 – geometricks/data_structure/internal/utils.hpp

```
1  #ifndef GEOMETRICKS_DATA_STRUCTURE_PRIMITIVES_INTERNAL_UTILS_HPP
2  #define GEOMETRICKS_DATA_STRUCTURE_PRIMITIVES_INTERNAL_UTILS_HPP
3
4  #include <stdint.h>
5
6  namespace geometricks {
7
8    namespace __detail__ {
9
10     uint64_t
11     next_power_of_2( uint64_t input ) {
12       --input;
13       input |= input >> 1;
14       input |= input >> 2;
15       input |= input >> 4;
16       input |= input >> 8;
17       input |= input >> 16;
18       input |= input >> 32;
19       ++input;
20       return input;
21     }
22
23     int64_t
24     next_power_of_2( int64_t input ) {
25       if( input < 0 ) return input;
26       union signed_unsigned {
27         int64_t sig;
28         uint64_t unsig;
29       };
30       signed_unsigned tmp = { input };
31       tmp.unsig = next_power_of_2( tmp.unsig );
32       return tmp.sig;
33     }
34
35     uint32_t
36     next_power_of_2( uint32_t input ) {
37       --input;
38       input |= input >> 1;
```

```
39        input |= input >> 2;
40        input |= input >> 4;
41        input |= input >> 8;
42        input |= input >> 16;
43        ++input;
44        return input;
45      }
46
47      int32_t
48      next_power_of_2( int32_t input ) {
49        if( input < 0 ) return input;
50        union signed_unsigned {
51          int32_t sig;
52          uint32_t unsig;
53        };
54        signed_unsigned tmp = { input };
55        tmp.unsig = next_power_of_2( tmp.unsig );
56        return tmp.sig;
57      }
58
59      uint16_t
60      next_power_of_2( uint16_t input ) {
61        --input;
62        input |= input >> 1;
63        input |= input >> 2;
64        input |= input >> 4;
65        input |= input >> 8;
66        ++input;
67        return input;
68      }
69
70      int16_t
71      next_power_of_2( int16_t input ) {
72        if( input < 0 ) return input;
73        union signed_unsigned {
74          int16_t sig;
75          uint16_t unsig;
76        };
77        signed_unsigned tmp = { input };
```

```
78        tmp.unsig = next_power_of_2( tmp.unsig );
79        return tmp.sig;
80      }
81
82      uint8_t
83      next_power_of_2( uint8_t input ) {
84        --input;
85        input |= input >> 1;
86        input |= input >> 2;
87        input |= input >> 4;
88        ++input;
89        return input;
90      }
91
92      int8_t
93      next_power_of_2( int8_t input ) {
94        if( input < 0 ) return input;
95        union signed_unsigned {
96          int8_t sig;
97          uint8_t unsig;
98        };
99        signed_unsigned tmp = { input };
100       tmp.unsig = next_power_of_2( tmp.unsig );
101       return tmp.sig;
102     }
103
104   }
105
106 }
107
108 #endif
```

Código B.8 – geometricks/data_structure/dimensional_traits.hpp

```
1 #ifndef GEOMETRICKS_DATA_STRUCTURE_MULTIDIMENSIONAL_DIMENSIONAL_TRAITS_HPP
2 #define GEOMETRICKS_DATA_STRUCTURE_MULTIDIMENSIONAL_DIMENSIONAL_TRAITS_HPP
3
4 //C++ stblib includes
5 #include <utility>
6 #include <tuple>
7 #include <array>
8
9 //Project includes
10 #include "geometricks/meta/utils.hpp"
11 #include "geometricks/meta/detect.hpp"
12 #include "geometricks/algorithm/absolute_difference.hpp"
13
14 namespace geometricks {
15
16     namespace dimension {
17
18         /**
19         * @brief struct that represents a specific dimension, with 0 being the first.
20         * @see @ref geometricks::dimension::get_t "get".
21         */
22         template< int Index >
23         struct dimension_t {};
24
25         template< int Index >
26         constexpr dimension_t<Index> dimension_v;
27
28         /**
29         * @brief Traits that, for a given type, output how many dimensions that type has.
30         * @tparam T The user defined type.
31         * @details Specialize this struct with your own type and expose a static constexpr int dimensions variable that
```

```
32      * how many dimensions there are in the type. This should
          be done so that user defined types can integrate with
          the library.
33      * Example:
34      * @code{.cpp}
35        namespace thirdparty {
36          struct data_t {
37            int first;
38            int second;
39          };
40        }
41        namespace geometricks::dimension {
42          template<>
43          struct dimensional_traits<thirdparty::data_t> {
44            static constexpr int dimensions = 2;
45          };
46        }
47      * @endcode
48      * @see @ref geometricks::dimension::get_t "geometricks::
          dimension::get".
49      */
50      template< typename T >
51      struct dimensional_traits;
52
53      template< typename... Types >
54      struct dimensional_traits< std::tuple<Types...> > {
55
56        static constexpr int dimensions = sizeof...( Types );
57
58      };
59
60      template< typename T, typename U >
61      struct dimensional_traits<std::pair<T, U>> {
62
63        static constexpr int dimensions = 2;
64
65      };
66
```

```
67        template< typename T, size_t N >
68        struct dimensional_traits<std::array<T,N>> {
69          static constexpr int dimensions = N;
70        };
71
72        namespace get_customization {
73
74          /**
75          * @brief Template struct to customize access to data
               type dimensions.
76          * @tparam T The type.
77          * @details Specialize this struct in case you don't have
                access to the type namespace in order to allow easy
               integration with the library.
78          * For that, expose static methods that take const T& as
               the first parameter and geometricks::dimension::
               dimension_t as the dimension.
79          * Example:
80          * @code{.cpp}
81            namespace thirdparty {
82              struct data_t {
83                int first;
84                int second;
85              };
86            }
87            namespace geometricks::dimension::get_customization {
88              template<>
89              struct get<thirdparty::data_t> {
90
91                //Could return a const int&. Returning a copy for
                     performance reasons.
92                static int _( const thirdparty::data_t& data,
                     geometricks::dimension::dimension_t<0> ) {
93                  return data.first;
94                }
95
96                //Could return a const int&. Returning a copy for
                     performance reasons.
97                static int _( const thirdparty::data_t& data,
```

```
                geometricks::dimension::dimension_t<1> ) {
98                return data.second;
99              }
100
101           };
102
103         }
104       * @endcode
105       * @see geometricks::dimension::dimensional_traits and
            geometricks::dimension::dimension_t.
106       */
107       template< typename T >
108       struct get;
109
110     }
111
112     /**
113     * @cond EXCLUDE_DOXYGEN
114     *
115     * Internal not to be documented
116     */
117     namespace __detail__ {
118
119       template< int I >
120       constexpr decltype( auto )
121       get( int );
122
123       template< int I >
124       constexpr decltype( auto )
125       get( int, dimension_t<I> );
126
127       template< int I, typename T >
128       constexpr decltype( auto )
129       __get__( T&& value, geometricks::meta::priority_tag<0> )
              -> decltype( get<I>( std::forward<T>( value ) ) ) {
130         return get<I>( std::forward<T>( value ) );
131       }
132
133       template< int I, typename T >
```

```
134         constexpr decltype( auto )
135         __get__( T&& value, geometricks::meta::priority_tag<1> )
                -> decltype( get( std::forward<T>( value ),
               dimension_v<I> ) ) {
136           return get( std::forward<T>( value ), dimension_v<I> )
                  ;
137         }
138
139         template< int I, typename T >
140         constexpr decltype( auto )
141         __get__( T&& value, geometricks::meta::priority_tag<2> )
                -> decltype( get_customization::get<std::decay_t<T
               >>::_( std::forward<T>( value ), dimension_v<I> ) ) {
142           return get_customization::get<std::decay_t<T>>::_( std
                  ::forward<T>( value ), dimension_v<I> );
143         }
144
145       }
146       /**
147       * @endcond
148       */
149
150       /**
151       * @brief struct that extracts the data in a given
             dimension from a type.
152       * @details The data is extracted according to the
             following protocol: first, it tries to access a
             specialization for the geometricks::dimension::
             get_customization::get
153       * template struct for the given type and call a static
             function named _ that accepts the data as the first
             parameter and a geometricks::dimension::dimension_t as
             the second parameter.
154       * If there is no specialization, it tries to access a
155       * get free function that accepts both the data and a
             geometricks::dimension::dimension_t for the specific
             dimension. Finally, if none of the alternatives can be
             done, it
156       * tries calling a get free function that takes an index
```

```
        representing the current dimension template argument
        and the data type as an argument. In case none of those
         functions are present,
157     * it is a compilation error.
158     * @note In case you have access to the data type namespace
        , you can just supply a free function taking a
        geometricks::dimension::dimension_t for each of the
        types dimension
159     * to allow the library to access your type.
160     * @see geometricks::dimension::get_customization::get for
        details on how to customize the type in case you don't
        have access to the type namespace.
161     */
162     struct get_t {
163
164       template< typename T, int I >
165       constexpr decltype( auto )
166       operator() ( T&& value, dimension_t<I> ) const {
167         return __detail__::__get__<I>( std::forward<T>( value
              ), geometricks::meta::priority_tag<2>{} );
168       }
169
170     };
171
172     constexpr get_t get = {};
173
174     template< typename T, int I >
175     using type_at = std::decay_t<decltype( get( std::declval<T
        >(), dimension_v<I> ) )>;
176
177     /**
178     * @todo Move this to another file and implement other
        distance functions.
179     */
180     struct euclidean_distance {
181
182     private:
183
184       template< typename T >
```

```
185         static auto
186         element_distance( const T& lhs, const T& rhs ) noexcept
               {
187           auto tmp = algorithm::absolute_difference( lhs, rhs );
188           return tmp * tmp;
189         }
190
191         template< size_t... Index >
192         auto
193         distance_impl( const auto& lhs, const auto& rhs, std::
               index_sequence<Index...> ) const noexcept {
194           return ( element_distance( dimension::get( lhs,
                 dimension_t<Index>{} ), dimension::get( rhs,
                 dimension_t<Index>{} ) ) + ... );
195         }
196
197       public:
198
199         template< typename T, typename U, int Index >
200         auto
201         operator()( const T& element, const U& stored,
               dimension_t<Index> ) const noexcept {
202           return element_distance( dimension::get( element,
                 dimension_t<Index>{} ), stored );
203         }
204
205         template< typename T, int Index >
206         auto
207         operator()( const T& element, const T& stored,
               dimension_t<Index> ) const noexcept {
208           return element_distance( dimension::get( element,
                 dimension_t<Index>{} ), dimension::get( stored,
                 dimension_t<Index>{} ) );
209         }
210
211         template< typename T >
212         auto
213         operator()( const T& lhs, const T& rhs ) const noexcept
               {
```

```
214             return distance_impl( lhs, rhs, std::
                    make_index_sequence<dimensional_traits<T>::
                    dimensions>{} );
215           }
216
217         };
218
219       } //namespace dimension
220
221       /**
222       * @cond EXCLUDE_DOXYGEN
223       *
224       * Internal not to be documented
225       */
226       namespace __detail__ {
227
228         template< typename Functor, typename T, typename U,
                    typename Dimension >
229         using compare_dimension = decltype( std::declval<Functor
                    >()( std::declval<T>(), std::declval<U>(), std::declval
                    <Dimension>() ) );
230
231         template< typename Functor, typename T, typename U >
232         using compare_value = decltype( std::declval<Functor>()(
                    std::declval<T>(), std::declval<U>() ) );
233
234         template< typename Functor, typename T, typename U, int I
                    >
235         constexpr bool has_dimension_compare = meta::
                    is_valid_expression_v<compare_dimension, Functor, T, U,
                     dimension::dimension_t<I>>;
236
237         template< typename Functor, typename T, typename U >
238         constexpr bool has_value_compare = meta::
                    is_valid_expression_v<compare_value, Functor, T, U>;
239
240       }
241       /**
242       * @endcond
```

```
243         */
244
245
246
247 } //namespace geometricks
248
249 #endif //
        GEOMETRICKS_DATA_STRUCTURE_MULTIDIMENSIONAL_DIMENSIONAL_TRAITS_HPP
```

Código B.9 – geometricks/data_structure/kd_tree.hpp

```
1  #ifndef GEOMETRICKS_DATA_STRUCTURE_KD_TREE_HPP
2  #define GEOMETRICKS_DATA_STRUCTURE_KD_TREE_HPP
3
4  //C++ stdlib includes
5  #include <functional>
6  #include <type_traits>
7  #include <algorithm>
8  #include <queue>
9  #include <vector>
10
11 //Project includes
12 #include "dimensional_traits.hpp"
13 #include "geometricks/meta/utils.hpp"
14 #include "geometricks/memory/allocator.hpp"
15 #include "internal/small_vector.hpp"
16
17 /**
18 * @file Implements a cache friendly kd tree stored as an array.
19 */
20
21 namespace geometricks {
22
23   /**
24   * @brief Cache friendly kd tree data structure
25   * @tparam T The stored data type.
26   * @tparam Compare Function that compares all the different
        data types stored in each dimension of the data so we can
        build the tree.
27   * If the stored data type T is a std::tuple<int, std::string,
        float>, the function should be able to compare ( int, int )
        , ( std::string, std::string ),
28   * ( float, float ) so we can work on all different dimensions.
29   * @details This kd tree is stored as an array in memory. This
        gives better cache locality than node based kd trees. The
        elements are stored in the nodes.
30   * Since it is extremely hard to balance a kd tree and it hurts
         performance to build a new one in each element insertion,
        insertion opperations are not allowed.
```

```
31  * @see geometricks::dimension::dimensional_traits and @ref
       geometricks::dimension::get_t "geometricks::dimension::get"
       for a guide on how to use this struct with user defined
       types.
32  * @see https://en.wikipedia.org/wiki/K-d_tree for a quick
       reference on kd tree.
33  * @todo Static assert on compare so we know it can sort in all
       dimensions.
34  * @todo noexcept and constexpr anotations.
35  * @todo Add threshold neighbors to find all elements below
       threshold distance to efficiently implement collision
       detection algorithms. Maybe?
36  */
37  template< typename T,
38            typename Compare = std::less<> >
39  struct kd_tree : private Compare {
40
41  private:
42
43    struct __heap_compare__ {
44      template< typename DistanceType >
45      constexpr bool operator()( const std::pair<T*,
          DistanceType>& lhs, const std::pair<T*, DistanceType>&
          rhs ) const noexcept {
46        return lhs.second < rhs.second;
47      }
48    };
49
50  public:
51
52    //Constructor
53
54    /**
55    * @brief Constructs a kd tree with a range of elements
56    * @param begin Iterator to first element of the input range.
57    * @param end Iterator to the last element of the input range
         or sentinel value.
58    * @param comp Compare function to use for the kd tree.
        Should be able to sort objects in different dimensions.
```

```
    If not supplied, default constructs it.
59      * @param alloc Memory allocator to use. Defaults to the
          default allocator. See also geometricks::allocator.
60      * @pre If Sentinel is an iterator, first < last. Else,
          eventually first != last compares false.
61      * @details Constructs a kd tree with the data supplied by
          the range [ begin, end ).
62      *
63      * @note Complexity: @b O(n log n)
64      * @see https://en.wikipedia.org/wiki/K-d_tree#Complexity
65      * @todo Supply paper on kd tree construction.
66      */
67      template< typename InputIterator, typename Sentinel >
68      kd_tree( InputIterator begin, Sentinel end, Compare comp =
          Compare{}, geometricks::allocator alloc = geometricks::
          allocator{} ): Compare( comp ),
69
70
```

```
71
72          __construct_kd_tree__<0>( begin, end, 0, m_size );
73      }
74
75      /**
```

```
76      * @brief Constructs a kd tree with a range of elements
77      * @param begin Iterator to first element of the input range.
78      * @param end Iterator to the last element of the input range
            or sentinel value.
79      * @param comp Placeholder used to call the default
            constructor for the Compare template parameter. See also
            geometricks::default_compare_t.
80      * @param alloc Memory allocator to use. Defaults to the
            default allocator. See also geometricks::allocator.
81      * @pre If Sentinel is an iterator, first < last. Else,
            eventually first != last compares false.
82      * @details Constructs a kd tree with the data supplied by
            the range [ begin, end ).
83      *
84      * @note Complexity: @b O(n log n)
85      * @see https://en.wikipedia.org/wiki/K-d_tree#Complexity
86      * @todo Supply paper on kd tree construction.
87      */
88     template< typename InputIterator, typename Sentinel >
89     kd_tree( InputIterator begin, Sentinel end, geometricks::
           default_compare_t comp, geometricks::allocator alloc =
           geometricks::allocator{} ):    m_allocator( alloc ),
90
```

```
91
92          ( void ) comp; //Silence warnings and errors.
93          __construct_kd_tree__<0>( begin, end, 0, m_size );
94      }
95
96      //Copy constructor
97
```

```
98      /**
99      * @brief Copy constructs a kd tree.
100     * @param rhs Right hand side of the copy operation.
101     * @param alloc Memory allocator to use. Defaults to the
            default allocator. See also geometricks::allocator
102     * @details Performs a deep copy of the right hand side
            parameter.
103     * @note Complexity: @b O(n)
104     */
105     kd_tree( const kd_tree& rhs, geometricks::allocator alloc =
            geometricks::allocator{} ):    Compare( rhs ),
106
107
108
```

```
109         std::copy( rhs.m_data_array, rhs.m_data_array + m_size,
                m_data_array );
110     }
111
112     //Move constructor
113
114     /**
115     * @brief Move constructs a kd tree.
116     * @param rhs Right hand side of the move operation.
117     * @post Invalidates rhs. Any use of rhs after move is an
            error.
118     * @details Moves the data from rhs into this.
119     * @note Complexity: @b O(1)
120     */
121     kd_tree( kd_tree&& rhs ): Compare( std::move( rhs ) ),
122                                        m_allocator( rhs.
                                               m_allocator ),
```

```
123                                                    m_size( rhs.m_size ),
124                                                    m_data_array( rhs.
                                                           m_data_array ) {
125       rhs.m_data_array = nullptr;
126     }
127
128     //Copy assignment
129
130     /**
131     * @brief Copy assigns a kd tree.
132     * @param rhs Right hand side of the copy operation.
133     * @details Performs a deep copy of the right hand side
            parameter. Destroys the previous kd tree and allocates
            memory to construct rhs into this.
134     * @note Complexity: @b O(n)
135     * @todo Change this method so we only allocate if the buffer
            isn't large enough.
136     * @todo Maybe we can get the strong exception guarantee here
            ...
137     */
138     kd_tree& operator=( const kd_tree& rhs ) {
139       if( &rhs != this ) {
140         //TODO: optimize to only allocate new buffer in case old
                capacity wasn't enough.
141         //TODO: exception guarantee.
142         Compare::operator=( rhs );
143         T* new_buff = ( T* ) m_allocator.allocate( sizeof( T ) *
                rhs.m_size );
144         std::copy( rhs.m_data_array, rhs.m_data_array + rhs.
                m_size, new_buff );
145         __destroy__();
146         m_data_array = new_buff;
147         m_size = rhs.m_size;
148       }
149       return *this;
150     }
151
152     //Move assignment
153
```

```
154     /**
155     * @brief Move assigns a kd tree.
156     * @param rhs Right hand side of the move operation.
157     * @post Invalidates rhs. Any use of rhs after move is an
            error.
158     * @details Moves the data from rhs into this.
159     * @note Complexity: @b O(1)
160     */
161     kd_tree& operator=( kd_tree&& rhs ) {
162       if( &rhs != this ) {
163         Compare::operator=( std::move( rhs ) );
164         __destroy__();
165         m_data_array = rhs.m_data_array;
166         m_size = rhs.m_size;
167         m_allocator = rhs.m_allocator;
168         rhs.m_data_array = nullptr;
169       }
170       return *this;
171     }
172
173     ~kd_tree() {
174       __destroy__();
175     }
176
177     /**
178     * @brief Finds the nearest neighbor of an input point.
179     * @param point The input point to query.
180     * @param f Point distance function object. Should be able to
            compare 2 points and return a size type as well as
181     * compare 2 points in a specific dimension and return a size
            type with the following signature: operator()( const T&
            left, T& right, dimension::dimension_t<Index> ) const
            noexcept.
182     * Also, distance( point1, point2 ) should be equal to
            distance( point2, point1 ).
183     * @details Computes the nearest neighbor of a given input
            point given the distance function. The default distance
            is the euclidean distance of the points without computing
184     * the square root to save on efficiency, since if sqrt(
```

```
        euclid_distance_no_sqrt_root(a, b) <
        euclid_distance_no_sqrt_root(a, c) ),
        euclid_distance_no_sqrt_root(a, b) <
        euclid_distance_no_sqrt_root(a, c).
185     *
186     * Example:
187     * @code{.cpp}
188     * std::vector<std::tuple<int, int, int>> input_vector;
189       ...
190       geometricks::kd_tree<std::tuple<int, int, int>> tree{
             input_vector.begin(), input_vector.end() };
191       struct manhattan_distance_t {
192         template< typename T, int N >
193         size_t operator()( const T& lhs, const T& rhs, dimension
              ::dimension_t<N> ) {
194           return geometricks::algorithm::absolute_difference(
                geometricks::dimension::get( lhs, dimension::
                dimension_v<N> ), geometricks::dimension::get( rhs,
                 dimension::dimension_v<N> ) );
195         }
196         template< typename T >
197         size_t operator()( const T& lhs, const T& rhs ) {
198           return distance_impl( lhs, rhs, std::
                make_index_sequence<dimension::dimensional_traits<T
                >::dimensions>() );
199         }
200         template< typename T, int... I >
201         size_t distance_impl( const T& lhs, const T& rhs, std::
              index_sequence<I...> ) {
202           return ( this->operator()( lhs, rhs, dimension_v<I> )
                + ... );
203         }
204       };
205       auto [nearest, distance] = tree.nearest_neighbor( std::
             make_tuple( 10, 10, 10 ), manhattan_distance_t{} );
206     * @endcode
207     * @todo Add references.
208     * @todo Add complexity.
209     * @todo Allow searching for threshold on nearest neighbor.
```

```
        Could be useful for code like collision detection.
210     */
211     template< typename DistanceFunction = dimension::
            euclidean_distance >
212     auto
213     nearest_neighbor( const T& point, DistanceFunction f =
            DistanceFunction{} ) const noexcept {
214       using distance_t = std::decay_t<decltype(f( std::declval<T
              >(), std::declval<T>() ))>;
215       distance_t best = meta::numeric_limits<distance_t>::max();
216       T* closest = nullptr;
217       __nearest_neighbor_impl__<0>( point, __root__(), &closest,
              best, f );
218       return std::pair<const T&, distance_t>( *closest, best );
219     }
220
221     /**
222     * @brief Finds the k nearest neighbors of an input point and
            returns a vector containing them and their distances.
223     * @param point The input point to query.
224     * @param K the number of desired output points.
225     * @param f Point distance function object. Should be able to
            compare 2 points and return a size type as well as
226     * compare 2 points in a specific dimension and return a size
            type with the following signature: operator()( const T&
            left, T& right, dimension::dimension_t<Index> ) const
            noexcept.
227     * Also, distance( point1, point2 ) should be equal to
            distance( point2, point1 ).
228     * @return A vector containing the output points as well as
            the distance calculated from the input point.
229     * @details Computes the k nearest neighbor of a given input
            point given the distance function. The default distance
            is the euclidean distance of the points without computing
230     * the square root to save on efficiency, since if sqrt(
            euclid_distance_no_sqrt_root(a, b) <
            euclid_distance_no_sqrt_root(a, c) ),
            euclid_distance_no_sqrt_root(a, b) <
            euclid_distance_no_sqrt_root(a, c).
```

```
231     * The points are returned in ascending order.
232     * Example:
233     * @code{.cpp}
234     * std::vector<std::tuple<int, int, int>> input_vector;
235       ...
236       geometricks::kd_tree<std::tuple<int, int, int>> tree{
             input_vector.begin(), input_vector.end() };
237       struct manhattan_distance_t {
238         template< typename T, int N >
239         size_t operator()( const T& lhs, const T& rhs, dimension
              ::dimension_t<N> ) {
240           return geometricks::algorithm::absolute_difference(
                 geometricks::dimension::get( lhs, dimension::
                 dimension_v<N> ), geometricks::dimension::get( rhs,
                  dimension::dimension_v<N> ) );
241         }
242         template< typename T >
243         size_t operator()( const T& lhs, const T& rhs ) {
244           return distance_impl( lhs, rhs, std::
                 make_index_sequence<dimension::dimensional_traits<T
                 >::dimensions>() );
245         }
246         template< typename T, int... I >
247         size_t distance_impl( const T& lhs, const T& rhs, std::
              index_sequence<I...> ) {
248           return ( this->operator()( lhs, rhs, dimension_v<I> )
                + ... );
249         }
250       };
251       auto output_vector = tree.k_nearest_neighbor( std::
             make_tuple( 10, 10, 10 ), 4, manhattan_distance_t{} );
             //output_vector now contains the 4 nearest neighbors of
              [10, 10, 10].
252     * @endcode
253     * @todo Add references.
254     * @todo Add complexity.
255     * @todo Allow searching for threshold on nearest neighbor.
           Could be useful for code like collision detection.
256     * @todo Improve performance by using a stack allocated
```

```
        vector as the max heeap, only fallbacking to the heap in
        case of a big K.
257     * @todo Allow alternative version of this function to
        receive the number of neighbors as a template parameter.
        Could be useful with a stack allocated vector.
258     * @todo Make a new version of this function that doesn't
        require an output_col as a parameter but simply returns a
         vector.
259     */
260     template< typename DistanceFunction = dimension::
        euclidean_distance >
261     auto
262     k_nearest_neighbor( const T& point, uint32_t K,
        DistanceFunction f = DistanceFunction{} ) ->
263     std::vector<std::pair<T, std::decay_t<decltype(f( std::
        declval<T>(), std::declval<T>() ))>>> {
264       using distance_t = std::decay_t<decltype(f( std::declval<T
          >(), std::declval<T>() ))>;
265       std::vector<std::pair<T, distance_t>> output_col;
266       output_col.reserve( K );
267       std::priority_queue<std::pair<T*, distance_t>,
          small_vector<std::pair<T*, distance_t>, 11>,
          __heap_compare__> max_heap;
268       __k_nearest_neighbor_impl__<0, DistanceFunction,
          distance_t>( point, __root__(), K, max_heap, f );
269       while( !max_heap.empty() ) {
270         auto& element = max_heap.top();
271         meta::add_element( std::make_pair( *element.first,
            element.second ), output_col );
272         max_heap.pop();
273       }
274       std::reverse( output_col.begin(), output_col.end() );
275       return output_col;
276     }
277
278     /**
279     * @brief Performs a range query on the collection.
280     * @param min_point Data containing the minimum values of the
         query.
```

```
281     * @param max_point Data containing the maximum values of the
            query.
282     * @return Vector containing all points in range.
283     * @details Computes and gathers all given points that lie
            within the region min_point and max_point and outputs
            them in a vector.
284     * Since the majority of the time is spent searching the tree
            for the output points, the algorithm first preprocess
            the input data so that there is no need for a
            precondition
285     * that the minimum point contains all the minimum values.
            Instead, the algorithm sorts both points before
            processing.
286     * Example:
287     * @code{.cpp}
288     std::vector<std::tuple<int, int, int>> input_vector;
289     ...
290     geometricks::kd_tree<std::tuple<int, int, int>> tree(
            input_vector.begin(), input_vector.end() );
291     auto output_vector = tree.range_search( std::make_tuple( 0,
            50, 300 ), std::make_tuple( 57, 51, 500 ) ); //
            output_vector now contains all points between [0-57,
            50-51, 300-500] from tree.
292     @endcode
293     * @todo Allow the user to input don't care values into the
            minimum and maximum point. Would need a new data
            structure for that.
294     */
295     std::vector<T>
296     range_search( T min_point, T max_point ) {
297       std::vector<T> output_col;
298       __organize_data__( min_point, max_point, std::
              make_index_sequence<DATA_DIMENSIONS>() );
299       __range_search_impl__<0>( min_point, max_point, __root__()
              , output_col );
300       return output_col;
301     }
302
303   private:
```

```
304
305     geometricks::allocator m_allocator;
306
307     int32_t m_size;
308
309     T* m_data_array;
310
311     static constexpr int DATA_DIMENSIONS = dimension::
            dimensional_traits<T>::dimensions;
312
313     struct node_t {
314
315       int32_t m_index;
316
317       int32_t m_block_size;
318
319       operator bool() const {
320         return m_block_size;
321       }
322
323     };
324
325     void
326     __destroy__() {
327       if( m_data_array != nullptr ) {
328         for( int32_t i = 0; i < m_size; ++i ) {
329           m_data_array[ i ].~T();
330         }
331         m_allocator.deallocate( m_data_array );
332       }
333     }
334
335     template< int Dimension, typename DistanceFunction, typename
            DistanceType >
336     void
337     __nearest_neighbor_impl__( const T& point, const node_t&
            cur_node, T** closest, DistanceType& best_distance,
            DistanceFunction f ) const {
338       constexpr size_t NextDimension = ( Dimension + 1 ) %
```

```
        DATA_DIMENSIONS;
339     auto compare_function = [this]( const T& left, const T&
          right ) {
340       return Compare::operator()( dimension::get( left,
            dimension::dimension_v<Dimension> ), dimension::get(
            right, dimension::dimension_v<Dimension> ) );
341     };
342     auto distance_function = [ &f ]( auto&& lhs, auto&& rhs )
          {
343       constexpr int I = Dimension;
344       if constexpr( __detail__::has_dimension_compare<
            DistanceFunction, T, T, I> ) {
345         return f( std::forward<decltype( lhs )>( lhs ), std::
              forward<decltype( rhs )>( rhs ), dimension::
              dimension_v<I> );
346       }
347       else if constexpr( __detail__::has_dimension_compare<
            DistanceFunction, T, dimension::type_at<T, I>, I> ) {
348         return f( std::forward<decltype( lhs )>( lhs ),
              dimension::get( std::forward<decltype( rhs )>( rhs
              ), dimension::dimension_v<I> ), dimension::
              dimension_v<I> );
349       }
350       else if constexpr( __detail__::has_dimension_compare<
            DistanceFunction, dimension::type_at<T, I>, T, I> ) {
351         return f( dimension::get( std::forward<decltype( lhs )
              >( lhs ), dimension::dimension_v<I> ), std::forward
              <decltype( rhs )>( rhs ), dimension::dimension_v<I>
               );
352       }
353       else if constexpr( __detail__::has_dimension_compare<
            DistanceFunction, dimension::type_at<T, I>, dimension
            ::type_at<T, I>, I> ) {
354         return f( dimension::get( std::forward<decltype( lhs )
              >( lhs ), dimension::dimension_v<I> ), dimension::
              get( std::forward<decltype( rhs )>( rhs ),
              dimension::dimension_v<I> ), dimension::dimension_v
              <I> );
355       }
```

```
356         else {
357           static_assert( __detail__::has_value_compare<
                DistanceFunction, dimension::type_at<T, I>,
                dimension::type_at<T, I>>, "Please supply a
                dimension compare, a value, value, dimension
                compare or a value compare." );
358           return f( dimension::get( std::forward<decltype( lhs )
                >( lhs ), dimension::dimension_v<I> ), dimension::
                get( std::forward<decltype( rhs )>( rhs ),
                dimension::dimension_v<I> ) );
359         }
360       };
361       if( compare_function( point, m_data_array[ cur_node.
            m_index ] ) ) {
362         //The point is to the left of the current axis.
363         //Recurse left...
364         auto left_child = __left_child__( cur_node );
365         if( left_child ) {
366           __nearest_neighbor_impl__<NextDimension>( point,
                left_child, closest, best_distance, f );
367         }
368         //Now we get the distance from the point to the current
              node.
369         auto distance = f( point, m_data_array[ cur_node.m_index
              ] );
370         //If the distance is better than our current best
              distance, update it.
371         if( distance < best_distance ) {
372           best_distance = distance;
373           *closest = &m_data_array[ cur_node.m_index ];
374         }
375         //If we have another branch to search...
376         auto right_child = __right_child__( cur_node );
377         if( right_child ) {
378           //Finally, check the distance to the hyperplane.
379           auto distance_to_hyperplane = distance_function( point
                , m_data_array[ cur_node.m_index ] );
380           if( distance_to_hyperplane < best_distance ) {
381             __nearest_neighbor_impl__<NextDimension>( point,
```

```
                right_child, closest, best_distance, f );
382           }
383         }
384
385       }
386       else {
387         //The point is to the right of the current axis.
388         //Recurse right..
389         auto right_child = __right_child__( cur_node );
390         if( right_child ) {
391           __nearest_neighbor_impl__<NextDimension>( point,
                right_child, closest, best_distance, f );
392         }
393         //Now we get the distance from the point to the current
              node.
394         auto distance = f( point, m_data_array[ cur_node.m_index
              ] );
395         //If the distance is better than our current best
              distance, update it.
396         if( distance < best_distance ) {
397           best_distance = distance;
398           *closest = &m_data_array[ cur_node.m_index ];
399         }
400         //If we have another branch to search...
401         auto left_child = __left_child__( cur_node );
402         if( left_child ) {
403           //Finally, check the distance to the hyperplane.
404           auto distance_to_hyperplane = distance_function( point
                , m_data_array[ cur_node.m_index ] );
405           if( distance_to_hyperplane < best_distance ) {
406             __nearest_neighbor_impl__<NextDimension>( point,
                  left_child, closest, best_distance, f );
407           }
408         }
409       }
410     }
411
412     template< int Dimension,
413               typename DistanceFunction,
```

```
414                 typename DistanceType >
415     void __k_nearest_neighbor_impl__( const T& point,
416                                       const node_t& node,
417                                       uint32_t K,
418                                       std::priority_queue<std::
                                             pair<T*, DistanceType>,
                                              small_vector<std::pair
                                             <T*, DistanceType>,
                                             11>, __heap_compare__>&
                                              max_heap,
419                                       DistanceFunction f ) {
420       constexpr size_t NextDimension = ( Dimension + 1 ) %
             DATA_DIMENSIONS;
421       auto compare_function = [this]( const T& left, const T&
             right ) {
422         return Compare::operator()( dimension::get( left,
               dimension::dimension_v<Dimension> ), dimension::get(
               right, dimension::dimension_v<Dimension> ) );
423       };
424
425       auto distance_function = [ &f ]( auto&& lhs, auto&& rhs )
             {
426         constexpr int I = Dimension;
427         if constexpr( __detail__::has_dimension_compare<
               DistanceFunction, T, T, I> ) {
428           return f( std::forward<decltype( lhs )>( lhs ), std::
                 forward<decltype( rhs )>( rhs ), dimension::
                 dimension_v<I> );
429         }
430         else if constexpr( __detail__::has_dimension_compare<
               DistanceFunction, T, dimension::type_at<T, I>, I> ) {
431           return f( std::forward<decltype( lhs )>( lhs ),
                 dimension::get( std::forward<decltype( rhs )>( rhs
                 ), dimension::dimension_v<I> ), dimension::
                 dimension_v<I> );
432         }
433         else if constexpr( __detail__::has_dimension_compare<
               DistanceFunction, dimension::type_at<T, I>, T, I> ) {
434           return f( dimension::get( std::forward<decltype( lhs )
```

```
            >( lhs ), dimension::dimension_v<I> ), std::forward
            <decltype( rhs )>( rhs ), dimension::dimension_v<I>
             );
435       }
436       else if constexpr( __detail__::has_dimension_compare<
            DistanceFunction, dimension::type_at<T, I>, dimension
            ::type_at<T, I>, I> ) {
437         return f( dimension::get( std::forward<decltype( lhs )
              >( lhs ), dimension::dimension_v<I> ), dimension::
              get( std::forward<decltype( rhs )>( rhs ),
              dimension::dimension_v<I> ), dimension::dimension_v
              <I> );
438       }
439       else {
440         static_assert( __detail__::has_value_compare<
              DistanceFunction, dimension::type_at<T, I>,
              dimension::type_at<T, I>>, "Please supply a
              dimension compare, a value, value, dimension
              compare or a value compare." );
441         return f( dimension::get( std::forward<decltype( lhs )
              >( lhs ), dimension::dimension_v<I> ), dimension::
              get( std::forward<decltype( rhs )>( rhs ),
              dimension::dimension_v<I> ) );
442       }
443     };
444     if( compare_function( point, m_data_array[ node.m_index ]
          ) ) {
445       auto left_child = __left_child__( node );
446       if( left_child ) {
447         __k_nearest_neighbor_impl__<NextDimension,
              DistanceFunction, DistanceType>( point, left_child,
               K, max_heap, f );
448       }
449       auto distance = f( point, m_data_array[ node.m_index ] )
            ;
450       auto heap_element = std::make_pair( &m_data_array[ node.
            m_index ], distance );
451       max_heap.push( heap_element );
452       if( ( uint32_t )max_heap.size() > K ) {
```

```
453             max_heap.pop();
454           }
455           auto right_child = __right_child__( node );
456           if( right_child ) {
457             auto distance_to_hyperplane = distance_function( point
                  , m_data_array[ node.m_index ] );
458             if( ( uint32_t )max_heap.size() < K ||
                  distance_to_hyperplane < max_heap.top().second ) {
459               __k_nearest_neighbor_impl__<NextDimension,
                    DistanceFunction, DistanceType>( point,
                    right_child, K, max_heap, f );
460             }
461           }
462         }
463         else {
464           auto right_child = __right_child__( node );
465           if( right_child ) {
466             __k_nearest_neighbor_impl__<NextDimension,
                  DistanceFunction, DistanceType>( point, right_child
                  , K, max_heap, f );
467           }
468           auto distance = f( point, m_data_array[ node.m_index ] )
                ;
469           auto heap_element = std::make_pair( &m_data_array[ node.
                m_index ], distance );
470           max_heap.push( heap_element );
471           if( ( uint32_t )max_heap.size() > K ) {
472             max_heap.pop();
473           }
474           auto left_child = __left_child__( node );
475           if( left_child ) {
476             auto distance_to_hyperplane = distance_function( point
                  , m_data_array[ node.m_index ] );
477             if( ( uint32_t )max_heap.size() < K ||
                  distance_to_hyperplane < max_heap.top().second ) {
478               __k_nearest_neighbor_impl__<NextDimension,
                    DistanceFunction, DistanceType>( point,
                    left_child, K, max_heap, f );
479             }
```

```
480         }
481       }
482     }
483
484     template< int Dimension, typename InputIterator, typename
          Sentinel >
485     void
486     __construct_kd_tree__( InputIterator begin, Sentinel end,
          int32_t startind_index, int32_t blocksize ) {
487       if( begin != end ) {
488         constexpr size_t NextDimension = ( Dimension + 1 ) %
            DATA_DIMENSIONS;
489         auto middle = begin;
490         int step = blocksize >> 1;
491         int32_t insert_index = startind_index + step;
492         std::advance( middle, step );
493         auto less_function = [this]( const T& left, const T&
            right ) {
494           return Compare::operator()( dimension::get( left,
              dimension::dimension_v<Dimension> ), dimension::get
              ( right, dimension::dimension_v<Dimension> ) );
495         };
496         std::nth_element( begin, middle ,end, less_function );
497         new ( &m_data_array[ insert_index ] ) T{ *middle };
498         __construct_kd_tree__<NextDimension>( begin, middle,
            startind_index, step );
499         std::advance( middle, 1 );
500         __construct_kd_tree__<NextDimension>( middle, end,
            insert_index + 1, blocksize - step - 1 );
501       }
502     }
503
504     template< typename _T >
505     constexpr bool
506     __compare__( const _T& first, const _T& second ) const {
507       return Compare::operator()( first, second );
508     }
509
510     node_t
```

```
511     __root__() const {
512       return { m_size >> 1, m_size };
513     }
514
515     static node_t
516     __left_child__( node_t node ) {
517       int32_t count_left = node.m_block_size >> 1;
518       int32_t index_left = ( node.m_index - ( count_left >> 1 )
            - ( count_left & 1 ) );
519       return { index_left, count_left };
520     }
521
522     static node_t
523     __right_child__( node_t node ) {
524       int32_t count_right = ( node.m_block_size >> 1 ) - !( node
            .m_block_size & 1 );
525       int32_t index_right = ( node.m_index + ( count_right >> 1
            ) + 1 );
526       return { index_right, count_right };
527     }
528
529     T&
530     __get__( node_t node ) {
531       return m_data_array[ node.m_index ];
532     }
533
534     const T&
535     __get__( node_t node ) const {
536       return m_data_array[ node.m_index ];
537     }
538
539     template< size_t... Is >
540     void
541     __organize_data__( T& first, T& second, std::index_sequence<
            Is...> ) {
542       ( __swap_if_greater__( dimension::get( first, dimension::
            dimension_v<Is> ), dimension::get( second, dimension::
            dimension_v<Is> ) ), ... );
543     }
```

```
544
545     template< typename DataType >
546     void
547     __swap_if_greater__( DataType& first, DataType& second ) {
548       if( !Compare::operator()( first, second ) ) {
549         using std::swap;
550         swap( first, second );
551       }
552     }
553
554     template< int CurrentDimension, typename Collection >
555     void
556     __range_search_impl__( const T& min_point, const T&
          max_point, node_t current_node, Collection&
          output_collection ) {
557       T& current_point = m_data_array[ current_node.m_index ];
558       constexpr int NextDimension = ( CurrentDimension + 1 ) %
            DATA_DIMENSIONS;
559       if( Compare::operator()( dimension::get( current_point,
            dimension::dimension_v<CurrentDimension> ), dimension::
            get( min_point, dimension::dimension_v<CurrentDimension
            > ) ) ) {
560         //If we're to the "left" side of the minimum value, we
              can discard the left children of this node since all
              of them would be on the left as well.
561         node_t next_node = __right_child__( current_node );
562         if( next_node ) {
563           __range_search_impl__<NextDimension>( min_point,
                max_point, next_node, output_collection );
564         }
565       }
566       else if( Compare::operator()( dimension::get( max_point,
            dimension::dimension_v<CurrentDimension> ), dimension::
            get( current_point, dimension::dimension_v<
            CurrentDimension> ) ) ) {
567         //If we're to the "right" side of the maximum value, we
              can discard the right children of this node since all
               of them would be on the right as well.
568         node_t next_node = __left_child__( current_node );
```

```
569         if( next_node ) {
570           __range_search_impl__<NextDimension>( min_point,
                max_point, next_node, output_collection );
571         }
572       }
573       else {
574         //If we're within the actual range, we have to check
              both children.
575         //Also, note that we only have to check other dimensions
              in the actual data if we're actually inside the
              range. If we're not in the range, it is not needed.
576         if( __is_inside_bounding_box__<CurrentDimension>(
              current_point, min_point, max_point) ) {
577           meta::add_element( m_data_array[ current_node.m_index
                ], output_collection );
578         }
579         node_t left_child = __left_child__( current_node );
580         if( left_child ) {
581           __range_search_impl__<NextDimension>( min_point,
                max_point, left_child, output_collection );
582         }
583         node_t right_child = __right_child__( current_node );
584         if( right_child ) {
585           __range_search_impl__<NextDimension>( min_point,
                max_point, right_child, output_collection );
586         }
587       }
588     }
589
590     template< int CurrentDimension >
591     constexpr bool
592     __is_inside_bounding_box__( const T& point, const T&
          min_point, const T& max_point ) {
593       return __is_inside_bounding_box_helper__<CurrentDimension
            >( point, min_point, max_point, std::
            make_index_sequence<DATA_DIMENSIONS>{} );
594     }
595
596     template< int CurrentDimension, size_t... Is >
```

```
597     constexpr bool
598     __is_inside_bounding_box_helper__( const T& point, const T&
            min_point, const T& max_point, std::index_sequence<Is...>
            ) {
599       return ( __is_inside_interval__<CurrentDimension, Is>(
              dimension::get( point, dimension::dimension_v<Is> ),
              dimension::get( min_point, dimension::dimension_v<Is> )
              , dimension::get( max_point, dimension::dimension_v<Is>
              ) ) && ... );
600     }
601
602     template< int CurrentDimension, int Index, typename DataType
            >
603     constexpr bool
604     __is_inside_interval__( const DataType& point, const
            DataType& min, const DataType& max ) {
605       if constexpr( CurrentDimension == Index ) {
606         return true;
607       }
608       else {
609         return point >= min && point <= max;
610       }
611     }
612
613   };
614
615 }
616
617 #endif //GEOMETRICKS_DATA_STRUCTURE_MULTIDIMENSIONAL_kd_tree_HPP
```

Código B.10 – geometricks/data_structure/quad_tree.hpp

```
1  #ifndef GEOMETRICKS_DATA_STRUCTURE_QUAD_TREE_HPP
2  #define GEOMETRICKS_DATA_STRUCTURE_QUAD_TREE_HPP
3
4  //C stdlib includes
5  #include <stdint.h>
6  #include <assert.h>
7
8  //C++ stdlib includesca
9  #include <type_traits>
10
11 //Project includes
12 #include "shapes.hpp"
13 #include "geometricks/memory/allocator.hpp"
14 #include "internal/small_vector.hpp"
15
16 namespace geometricks {
17
18   struct quad_tree_traits {
19     static constexpr int max_depth = 10;
20     static constexpr int leaf_size = 4;
21   };
22
23   template< typename T,
24             typename Traits = quad_tree_traits >
25   struct quad_tree {
26
27     quad_tree( geometricks::rectangle rect, geometricks::
          allocator alloc = geometricks::allocator{} ):
          m_root_rect( rect ),
28
29
```

```
30
31        m_nodes.push_back( __make_leaf__() );
32      }
33
34      bool insert( const T& element ) {
35        geometricks::rectangle bounding_rect = geometricks::
              make_rectangle( element );
36        if( !m_root_rect.contains( bounding_rect ) ) {
37          return false;
38        }
39        auto id = m_elements.push_back( std::make_pair( element,
              bounding_rect ) );
40        __insert__( { 0 }, m_root_rect, id, 0 );
41        return true;
42      }
43
44    private:
45
46      struct node_index_t {
47
48        int32_t m_value;
49
50        operator int32_t() const {
51          return m_value;
```

```
52        }
53
54      };
55
56      struct element_index_t {
57        int32_t m_value;
58
59        operator int32_t() const {
60          return m_value;
61        }
62
63      };
64
65      struct node {
66
67        //Indexes values in the quad tree. Last bit is used to
             represent if the value is a leaf or not.
68        struct index_t {
69
70          operator element_index_t() const {
71            return { static_cast<int32_t>( ( m_value & 0x7FFFFFFF
                 ) - 1 ) };
72          }
73
74          operator node_index_t() const {
75            return { static_cast<int32_t>( m_value ) };
76          }
77
78          uint32_t m_value;
79
80        };
81
82        bool
83        is_leaf() const {
84          return m_first_child.m_value >> 31;
85        }
86
87        node& operator=( element_index_t leaf_index ) {
88          m_first_child.m_value = 0x80000000 | ( leaf_index + 1 );
```

```
89          return *this;
90        }
91
92        node& operator=( node_index_t node_index ) {
93          m_first_child.m_value = node_index;
94          return *this;
95        }
96
97        index_t m_first_child;
98
99      };
100
101     //Since each element might be on more than one quadrant of
          the quad tree, we store 1 index to the element for each
          type it appears.
102     //TODO: think if we have to store the rectangle for each
          element. (Probably a yes).
103     struct element_t {
104
105       int32_t id;
106
107       int32_t next;
108
109     };
110
111     //TODO: SFINAE this to set default value in case leaf_size
          is not found.
112     static constexpr int LEAF_SIZE = Traits::leaf_size;
113
114     //TODO: SFINAE this to set default value in case max_depth
          is not found.
115     static constexpr int MAX_DEPTH = Traits::max_depth;
116
117     static constexpr typename node::index_t EMPTY_LEAF = { 0
          x80000000 };
118
119     geometricks::rectangle m_root_rect;
120
121     geometricks::small_vector<node, 1> m_nodes; //Index 0
```

```
      represents the root.
122
123   geometricks::small_vector<element_t, LEAF_SIZE>
        m_element_ref;
124
125   geometricks::small_vector<std::pair<T, geometricks::
        rectangle>, LEAF_SIZE> m_elements;
126
127   static node
128   __make_leaf__() {
129     return node{ EMPTY_LEAF };
130   }
131
132   struct __split_ret__ {
133     geometricks::rectangle first;
134     geometricks::rectangle second;
135     geometricks::rectangle third;
136     geometricks::rectangle fourth;
137   };
138
139   __split_ret__
140   __split_rect__( const geometricks::rectangle& rect ) {
141     auto half_width = rect.width / 2;
142     auto half_height = rect.height / 2;
143     return { geometricks::rectangle{ rect.x_left, rect.y_top,
          half_width, half_height },
144              geometricks::rectangle{ rect.x_left + half_width
                 + 1, rect.y_top, rect.width - half_width - 1,
                 half_height },
145              geometricks::rectangle{ rect.x_left, rect.y_top +
                  half_height + 1, half_width, rect.height -
                 half_height - 1 },
146              geometricks::rectangle{ rect.x_left + half_width
                 + 1, rect.y_top + half_height + 1, rect.width
                 - half_width - 1, rect.height - half_height -
                 1 } };
147   }
148
149   int
```

```
150     __count_leaf_objects__( int32_t index ) {
151       int ret = 0;
152       while( index != -1 ) {
153         index = m_element_ref[ index ].next;
154         ++ret;
155       }
156       return ret;
157     }
158
159     //TODO
160     void
161     __split_node__( node_index_t cur_node, const geometricks::
            rectangle& rect, int depth ) {
162       //Create 4 new nodes. Edit small vector class to allow
              insertion of more than 1 object at a time.
163       auto [ first, second, third, fourth ] = __split_rect__(
              rect );
164       element_index_t first_child = m_nodes[ cur_node ].
              m_first_child;
165       int32_t index = first_child;
166       node_index_t first_node = { m_nodes.push_back(
              __make_leaf__() ) };
167       m_nodes.push_back( __make_leaf__() );
168       m_nodes.push_back( __make_leaf__() );
169       m_nodes.push_back( __make_leaf__() );
170       assert( m_nodes[ cur_node ].is_leaf() );
171       m_nodes[ cur_node ] = first_node;
172       assert( !m_nodes[ cur_node ].is_leaf() );
173       while( index != -1 ) {
174         auto actual_id = m_element_ref[ index ].id;
175         geometricks::rectangle element_rect = m_elements[
                actual_id ].second;
176         if( geometricks::intersects_with( first, element_rect )
                ) {;
177           __insert__( first_node, first, actual_id, depth + 1 );
178         }
179         if( geometricks::intersects_with( second, element_rect )
                ) {
180           __insert__( { first_node + 1 }, second, actual_id,
```

```
            depth + 1 );
181       }
182       if( geometricks::intersects_with( third, element_rect )
            ) {
183         __insert__( { first_node + 2 }, third, actual_id,
              depth + 1 );
184       }
185       if( geometricks::intersects_with( fourth, element_rect )
            ) {
186         __insert__( { first_node + 3 }, fourth, actual_id,
              depth + 1 );
187       }
188       index = m_element_ref[ index ].next;
189     }
190   }
191
192   void
193   __insert__( node_index_t cur_node, const geometricks::
        rectangle& rect, int32_t object_id, int depth ) {
194     if( !m_nodes[ cur_node ].is_leaf() ) {
195       auto [ first, second, third, fourth ] = __split_rect__(
            rect );
196       node_index_t first_node_child = m_nodes[ cur_node ].
            m_first_child;
197       if( geometricks::intersects_with( first, m_elements[
            object_id ].second ) ) {
198         __insert__( first_node_child, first, object_id, depth
              + 1 );
199       }
200       if( geometricks::intersects_with( second, m_elements[
            object_id ].second ) ) {
201         __insert__( { first_node_child + 1 }, second,
              object_id, depth + 1 );
202       }
203       if( geometricks::intersects_with( third, m_elements[
            object_id ].second ) ) {
204         __insert__( { first_node_child + 2 }, third, object_id
              , depth + 1 );
205       }
```

```
206           if( geometricks::intersects_with( fourth, m_elements[
                  object_id ].second ) ) {
207             __insert__( { first_node_child + 3 }, fourth,
                  object_id, depth + 1 );
208           }
209         }
210         else {
211           element_index_t last_element_index = m_nodes[ cur_node
                  ].m_first_child;
212           element_t new_el{ object_id, last_element_index };
213           element_index_t element_id = { m_element_ref.push_back(
                  new_el ) };
214           m_nodes[ cur_node ] = element_id;
215           if( !( depth == MAX_DEPTH || rect.height < 3 || rect.
                  width < 3 ) ) {
216             auto count = __count_leaf_objects__( element_id );
217             if( count > LEAF_SIZE ) {
218               assert( m_nodes[ cur_node ].is_leaf() );
219               __split_node__( cur_node, rect, depth );
220               assert( !m_nodes[ cur_node ].is_leaf() );
221             }
222           }
223         }
224       }
225
226     };
227
228 } //namespace geometricks
229
230 #endif //GEOMETRICKS_DATA_STRUCTURE_QUAD_TREE_HPP
```

Código B.11 – geometricks/data_structure/shapes.hpp

```
1  #ifndef GEOMETRICKS_DATA_STRUCTURE_MULTIDIMENSIONAL_SHAPES_HPP
2  #define GEOMETRICKS_DATA_STRUCTURE_MULTIDIMENSIONAL_SHAPES_HPP
3
4  //C++ stblib includes
5  #include <iostream>
6  #include <stdint.h>
7
8  //Project includes
9  #include "geometricks/meta/utils.hpp"
10
11 namespace geometricks {
12
13   struct rectangle {
14
15     int32_t x_left;
16
17     int32_t y_top;
18
19     int32_t width;
20
21     int32_t height;
22
23     rectangle() = default;
24
25     constexpr rectangle( int32_t x_left_, int32_t y_top_,
          int32_t width_, int32_t height_ ):  x_left( x_left_ ),
26
27
```

```
28
29
30      rectangle( const rectangle& ) = default;
31
32      rectangle( rectangle&& ) = default;
33
34      rectangle& operator=( const rectangle& ) = default;
35
36      rectangle& operator=( rectangle&& ) = default;
37
38      int32_t xleft() const {
39        return x_left;
40      }
41
42      int32_t xright() const {
43        return x_left + width;
44      }
45
46      int32_t ytop() const {
47        return y_top;
48      }
49
50      int32_t ybot() const {
51        return y_top + height;
52      }
53
54      bool contains( const rectangle& other ) const {
```

```
55        return xleft() <= other.xleft() && xright() >= other.
              xright() && ytop() <= other.ytop() && ybot() >= other.
              ybot();
56      }
57
58    };
59
60    std::ostream& operator<<( std::ostream& os, const rectangle&
          rect ) {
61      os << "Rect at -> TL: (" << rect.x_left << ':' << rect.y_top
            << ") ";
62      os << "BR: (" << rect.x_left + rect.width << ':' << rect.
          y_top + rect.height << ") ";
63      return os;
64    }
65
66    constexpr bool intersects_with( const rectangle& lhs, const
          rectangle& rhs ) {
67      return  !( ( lhs.x_left > ( rhs.x_left + rhs.width ) ) || (
          rhs.x_left > ( lhs.x_left + lhs.width ) ) ||
68            ( lhs.y_top > ( rhs.y_top + rhs.height ) ) || ( rhs.
                y_top > ( lhs.y_top + lhs.height ) ) );
69    }
70
71    namespace rectangle_customization {
72
73      //Speciallize this struct with a static function called _
          that takes an object of your type and returns a rectangle
          .
74      template< typename T >
75      struct make_rectangle;
76
77    }
78
79    namespace __detail__ {
80
81      template< int I, typename T >
82      int get( T&& );
83
```

```
84      template< typename T >
85      constexpr auto
86      __make_rect__( const T& value, meta::priority_tag<0> ) ->
           decltype( rectangle( value[0], value[1], value[2], value
           [3] ) ) {
87        return rectangle( value[0], value[1], value[2], value[3] )
             ;
88      }
89
90      template< typename T >
91      constexpr auto
92      __make_rect__( const T& value, meta::priority_tag<1> ) ->
           decltype( rectangle( get<0>( value ), get<1>( value ),
           get<2>( value ), get<3>( value ) ) ) {
93        return rectangle( get<0>( value ), get<1>( value ), get
             <2>( value ), get<3>( value ) );
94      }
95
96      template< typename T >
97      constexpr auto
98      __make_rect__( const T& value, meta::priority_tag<2> ) ->
           std::enable_if_t<std::is_convertible_v<decltype( value.
           bounding_rect() ), rectangle>, decltype( value.
           bounding_rect() )> {
99        return static_cast<rectangle>( value.bounding_rect() );
100     }
101
102     template< typename T >
103     constexpr auto
104     __make_rect__( const T& value, meta::priority_tag<3> ) ->
           std::enable_if_t<std::is_convertible_v<decltype( value ),
            rectangle>, rectangle> {
105       return static_cast<rectangle>( value );
106     }
107
108     template< typename T >
109     constexpr auto
110     __make_rect__( const T& value, meta::priority_tag<4> ) ->
           decltype( rectangle_customization::make_rectangle<T>::_(
```

```
        value ) ) {
111       return rectangle_customization::make_rectangle<T>::_(
            value );
112     }
113
114   }
115
116   inline constexpr auto
117   make_rectangle = []( const auto& value ) -> decltype(
        __detail__::__make_rect__( value, meta::priority_tag<4>{} )
        ) {
118     return __detail__::__make_rect__( value, meta::priority_tag
          <4>{} );
119   };
120
121 }
122
123 #endif //GEOMETRICKS_DATA_STRUCTURE_MULTIDIMENSIONAL_SHAPES_HPP
```

Código B.12 – geometricks/memory/allocator/malloc_allocator.hpp

```
1  #ifndef GEOMETRICKS_MEMORY_ALLOCATOR_MALLOC_ALLOCATOR_HPP
2  #define GEOMETRICKS_MEMORY_ALLOCATOR_MALLOC_ALLOCATOR_HPP
3
4  //C stdlib includes
5  #include <stdint.h>
6  #include <stddef.h>
7  #include <stdlib.h>
8
9  namespace geometricks {
10
11   namespace memory {
12
13     struct malloc_allocator_t final {} malloc_allocator;
14
15     void*
16     allocate( malloc_allocator_t&, size_t sz ) {
17       return malloc( sz );
18     }
19
20     void*
21     reallocate( malloc_allocator_t&, void* ptr, size_t sz ) {
22       return realloc( ptr, sz );
23     }
24
25     void
26     deallocate( malloc_allocator_t&, void* ptr ) {
27       free( ptr );
28     }
29
30   }
31
32 }
33
34 #endif //GEOMETRICKS_MEMORY_ALLOCATOR_MALLOC_ALLOCATOR_HPP
```

Código B.13 – geometricks/memory/allocator/new_allocator.hpp

```
1  #ifndef GEOMETRICKS_MEMORY_ALLOCATOR_NEW_ALLOCATOR_HPP
2  #define GEOMETRICKS_MEMORY_ALLOCATOR_NEW_ALLOCATOR_HPP
3
4  #include <new>
5
6  namespace geometricks {
7
8    namespace memory {
9
10     struct new_allocator_t final {} new_allocator;
11
12     void*
13     allocate( new_allocator_t&, size_t sz ) {
14       return ::operator new( sz );
15     }
16
17     void
18     deallocate( new_allocator_t&, void* ptr ) {
19       ::operator delete( ptr );
20     }
21
22   }
23
24 }
25
26 #endif
```

Código B.14 – geometricks/memory/allocator/pool_allocator.hpp

```
1  #ifndef GEOMETRICKS_MEMORY_ALLOCATOR_POOL_ALLOCATOR_HPP
2  #define GEOMETRICKS_MEMORY_ALLOCATOR_POOL_ALLOCATOR_HPP
3
4  #include <type_traits>
5  #include <tuple>
6  #include <cstddef>
7
8  namespace geometricks {
9
10   namespace memory {
11
12     template< int PoolElementSize, int Size = 512, int Align =
           alignof( max_align_t ) >
13     class pool_allocator {
14
15       struct free_list_element {
16         free_list_element* next;
17       };
18
19       union free_t {
20         free_list_element* first_free;
21         std::aligned_storage_t<PoolElementSize, Align> _; //
              Force size and alignment.
22       };
23
24       struct block {
25
26         block* next_block = nullptr;
27
28         std::aligned_storage_t<sizeof( free_t ), alignof( free_t
              )> data[ Size ];
29
30         block() {
31           free_t* first = ( free_t* )&data;
32           free_t* last = first + ( Size - 1 );
33           while( first != last ) {
34             first->first_free = ( free_list_element* )( first +
                  1 );
```

```
35            ++first;
36          }
37          first->first_free = nullptr;
38        }
39
40      };
41
42      block* first_block;
43
44      free_t data;
45
46    public:
47
48      pool_allocator(): first_block( new block{} ), data{
           static_cast<free_list_element*>( static_cast<void*>( &
           first_block->data ) ) } {
49      }
50
51      ~pool_allocator() {
52        while( first_block ) {
53          block* to_delete = first_block;
54          first_block = first_block->next_block;
55          delete to_delete;
56        }
57      }
58
59      void* allocate( size_t sz ) {
60        ( void ) sz;
61        if( data.first_free != nullptr ) {
62          void* ret = (void*) data.first_free;
63          data.first_free = data.first_free->next;
64          return ret;
65        }
66        else {
67          block* next_first_block = new block();
68          next_first_block->next_block = first_block;
69          first_block = next_first_block;
70          data.first_free = static_cast<free_list_element*>(
              static_cast<void*>( &first_block->data ) );
```

```
71            void* ret = (void*) data.first_free;
72            data.first_free = data.first_free->next;
73            return ret;
74          }
75        }
76
77        void deallocate( void* ptr ) {
78          free_list_element* insert_ptr = ( free_list_element* )
              ptr;
79          insert_ptr->next = data.first_free;
80          data.first_free = insert_ptr;
81        }
82
83      };
84
85      template< int Sz, int Align >
86      struct size_align_pair {
87        static constexpr int size = Sz;
88        static constexpr int align = Align;
89      };
90
91      template< typename T >
92      struct get_pool_allocator;
93
94      template< int Sz, int Align >
95      struct get_pool_allocator<size_align_pair<Sz, Align>> {
96
97      };
98
99      template< typename T >
100     struct type_t {
101
102     };
103
104     template< typename T >
105     constexpr auto type = type_t<T>{};
106
107     template< typename... >
108     struct type_list {};
```

```
109
110     template< typename T, typename... Ts >
111     constexpr bool is_in( type_t<T>, type_list< type_t<Ts>... >
           ) {
112       return ( std::is_same_v<T, Ts> || ... );
113     }
114
115     template< typename T, typename... Ts >
116     constexpr auto append_unique_single( type_list< type_t<Ts
           >... > list , type_t<T> new_type ) {
117       if constexpr( is_in( new_type, list ) ) {
118         return list;
119       }
120       else {
121         return type_list< type_t<T>, type_t<Ts>... >{};
122       }
123     }
124
125     template< typename T, typename... Ts, typename... Tss >
126     constexpr auto append_unique( type_list< type_t<Tss>... >
           list , type_t<T> head, type_t<Ts>... tail ) {
127       constexpr auto appended = append_unique_single( list , head
             );
128       return append_unique( appended, tail... );
129     }
130
131     template< typename... T >
132     constexpr auto append_unique( type_list< type_t<T>... > list
           ) {
133       return list;
134     }
135
136     template< int BlockSize, typename... T >
137     struct multipool_allocator;  //This class is super awkward
           to use so far with node based strucutres that you can't
           know the node size without inspecting it beforehand.
138
139     template< int BlockSize, int... Sz, int... Align >
140     class multipool_allocator<BlockSize, size_align_pair<Sz,
```

```
     Align>...> {
141
142      std::tuple<pool_allocator<Sz, BlockSize, Align>...>
           m_allocators;
143
144    public:
145
146      void* allocate( size_t size, size_t align = alignof( std::
           max_align_t ) ) {
147        return __allocate_impl__( size, align, std::
             make_index_sequence<sizeof...(Sz)>() );
148      }
149
150      void deallocate( void* ptr, size_t size ) {
151        __deallocate_impl__( ptr, size, std::make_index_sequence
             <sizeof...(Sz)>() );
152      }
153
154    private:
155
156      template< size_t... Is >
157      void*
158      __allocate_impl__( size_t size, size_t align, std::
           index_sequence<Is...> ) {
159        void* ret = nullptr;
160        ( __allocate_impl_helper__( size, align, std::get<Is>(
             m_allocators ), &ret ), ... );
161        return ret;
162      }
163
164      template< int Sz_, int Align_ >
165      void
166      __allocate_impl_helper__( size_t size, size_t ,
           pool_allocator<Sz_, BlockSize, Align_>& pool, void**
           ptr ) {
167        if( size == Sz_ ) {
168          *ptr = pool.allocate( size );
169        }
170      }
```

```
171
172         template< size_t... Is >
173         void
174         __deallocate_impl__( void* ptr, size_t size, std::
               index_sequence<Is...> ) {
175           ( __deallocate_impl_helper__( ptr, size, std::get<Is>(
                 m_allocators ) ), ... );
176         }
177
178         template< int Sz_, int Align_ >
179         void
180         __deallocate_impl_helper__( void* ptr, size_t size,
               pool_allocator<Sz_, BlockSize, Align_>& pool ) {
181           if( size == Sz_ ) {
182             pool.deallocate( ptr );
183           }
184         }
185
186       };
187
188       template< int BlockSize, typename... Ts >
189       class multipool_allocator<BlockSize, const type_list<type_t<
             Ts>...>>: public multipool_allocator<BlockSize, Ts...> {
190
191       };
192
193       template< int BlockSize, typename... Ts >
194       auto make_multipool_allocator( type_t<Ts>... ) {
195         constexpr type_list<> list{};
196         constexpr auto appended = append_unique( list, type<
               size_align_pair<sizeof(Ts), alignof( std::max_align_t )
               >>... );
197         return multipool_allocator<BlockSize, decltype( appended )
               >{};
198       }
199
200     }
201
202 }
```

```
203
204 #endif
```

Código B.15 – geometricks/memory/allocator.hpp

```
1  #ifndef GEOMETRICKS_MEMORY_ALLOCATOR_HPP
2  #define GEOMETRICKS_MEMORY_ALLOCATOR_HPP
3
4  #include <type_traits>
5  #include <cstddef>
6
7  #include "allocator/malloc_allocator.hpp"
8  #include "geometricks/meta/detect.hpp"
9  #include "geometricks/meta/utils.hpp"
10
11 /**
12 * @file
13 * @brief Implements the allocator concepts used for the data
      structures of this project.
14 */
15
16 namespace geometricks {
17
18   struct allocator;
19
20   namespace memory {
21
22     namespace allocator_customization {
23
24       /**
25       * @brief Adaptor struct that should be specialized for
            types that can allocate and deallocate memory but don't
            conform to the interface of the library.
26       * @tparam T Template parameter that should be specialized.
27       * @details The specialization of this struct should
            contain at least 1 function to allocate memory and 1
            function to deallocate memory.
28       * It can also optionally contain a function to reallocate
            memory.
29       * Functions to allocate memory should comform to the
            following interface
30       * @code{.cpp}
31         struct my_allocator_t { ... };
```

```
32          namespace geometricks::memory::allocator_customization {
33            struct allocator<my_allocator_t> {
34
35              //Can be either this function in case of aligned
                    memory.
36              static void* allocate( my_alloc_t&, size_t, size_t )
                    ;
37
38              //Or this function in case of unaligned memory.
39              static void* allocate( my_alloc_t&, size_t );
40
41              ...
42
43            };
44          }
45        * @endcode
46
47        * Functions to deallocate memory should comform to the
             following interface
48        * @code{.cpp}
49          struct my_allocator_t { ... };
50          namespace geometricks::memory::allocator_customization {
51            struct allocator<my_allocator_t> {
52
53              //Can be either this function in case there is a
                    need to know the size of the element.
54              void* deallocate( my_alloc_t&, void*, size_t );
55
56              //Or this function otherwise.
57              void deallocate( my_alloc_t&, void* );
58
59              ...
60
61            };
62          }
63        * @endcode
64        */
65        template< typename T >
66        struct allocator;
```

```
67
68      }
69
70      /**
71      * @cond EXCLUDE_DOXYGEN
72      *
73      * Internal not to be documented
74      */
75      namespace __detail__ {
76
77        template< typename Allocator >
78        auto __do_allocate_unaligned__( Allocator& allocator,
            size_t sz, geometricks::meta::priority_tag<0> ) ->
            decltype( alloc( allocator, sz ) ) {
79          return alloc( allocator, sz );
80        }
81
82        template< typename Allocator >
83        auto __do_allocate_unaligned__( Allocator& alloc, size_t
            sz, geometricks::meta::priority_tag<1> ) -> decltype(
            allocate( alloc, sz ) ) {
84          return allocate( alloc, sz );
85        }
86
87        template< typename Allocator >
88        auto __do_allocate_unaligned__( Allocator& alloc, size_t
            sz, geometricks::meta::priority_tag<2> ) -> decltype(
            alloc.alloc( sz ) ) {
89          return alloc.alloc( sz );
90        }
91
92        template< typename Allocator >
93        auto __do_allocate_unaligned__( Allocator& alloc, size_t
            sz, geometricks::meta::priority_tag<3> ) -> decltype(
            alloc.allocate( sz ) ) {
94          return alloc.allocate( sz );
95        }
96
97        template< typename Allocator >
```

```
98      auto __do_allocate_unaligned__( Allocator& alloc, size_t
          sz, geometricks::meta::priority_tag<4> ) -> decltype(
          allocator_customization::allocator<Allocator>::allocate
          ( alloc, sz ) ) {
99        return allocator_customization::allocator<Allocator>::
            allocate( alloc, sz );
100     }
101
102     template< typename T >
103     using __try_allocate_unalign__ = decltype(
          __do_allocate_unaligned__( std::declval<T&>(), std::
          declval<size_t>(), std::declval<meta::priority_tag
          <4>>() ) );
104
105     template< typename T >
106     constexpr bool __can_allocate_unalign__ = meta::
          is_valid_expression_v<__try_allocate_unalign__, T>;
107
108     struct __test_allocate_unalign__ {
109       void* alloc( size_t );
110     };
111
112     static_assert( __can_allocate_unalign__<
          __test_allocate_unalign__> );
113
114     static_assert( !__can_allocate_unalign__<int> );
115
116     template< typename Allocator >
117     auto __do_allocate_aligned__( Allocator& allocator, size_t
          sz, size_t align, meta::priority_tag<0> ) -> decltype(
          alloc( allocator, sz, align ) ) {
118       return alloc( allocator, sz, align );
119     }
120
121     template< typename Allocator >
122     auto __do_allocate_aligned__( Allocator& alloc, size_t sz,
          size_t align, meta::priority_tag<1> ) -> decltype(
          allocate( alloc, sz, align ) ) {
123       return allocate( alloc, sz, align );
```

```
124       }
125
126       template< typename Allocator >
127       auto __do_allocate_aligned__( Allocator& alloc, size_t sz,
              size_t align, meta::priority_tag<2> ) -> decltype(
             alloc.alloc( sz, align ) ) {
128         return alloc.alloc( sz, align );
129       }
130
131       template< typename Allocator >
132       auto __do_allocate_aligned__( Allocator& alloc, size_t sz,
              size_t align, meta::priority_tag<3> ) -> decltype(
             alloc.allocate( sz, align ) ) {
133         return alloc.allocate( sz, align );
134       }
135
136       template< typename Allocator >
137       auto __do_allocate_aligned__( Allocator& alloc, size_t sz,
              size_t align, meta::priority_tag<4> ) -> decltype(
             allocator_customization::allocator<Allocator>::allocate
             ( alloc, sz, align ) ) {
138         return allocator_customization::allocator<Allocator>::
               allocate( alloc, sz, align );
139       }
140
141       template< typename T >
142       using __try_allocate_align__ = decltype(
             __do_allocate_aligned__( std::declval<T&>(), std::
             declval<size_t>(), std::declval<size_t>(), std::declval
             <meta::priority_tag<4>>() ) );
143
144       template< typename T >
145       constexpr bool __can_allocate_align__ = meta::
             is_valid_expression_v<__try_allocate_align__, T>;
146
147       struct __test_allocate_align__ {
148         void* alloc( size_t, size_t );
149       };
150
```

```
151     static_assert( __can_allocate_align__<
            __test_allocate_align__> );
152
153     static_assert( !__can_allocate_align__<
            __test_allocate_unalign__> );
154
155     template< typename Allocator >
156     auto __do_deallocate_unaligned__( Allocator& allocator,
            void* ptr, meta::priority_tag<0> ) -> decltype( dealloc
            ( allocator, ptr ) ) {
157       return dealloc( allocator, ptr );
158     }
159
160     template< typename Allocator >
161     auto __do_deallocate_unaligned__( Allocator& allocator,
            void* ptr, meta::priority_tag<1> ) -> decltype(
            deallocate( allocator, ptr ) ) {
162       return deallocate( allocator, ptr );
163     }
164
165     template< typename Allocator >
166     auto __do_deallocate_unaligned__( Allocator& allocator,
            void* ptr, meta::priority_tag<2> ) -> decltype(
            allocator.dealloc( ptr ) ) {
167       return allocator.dealloc( ptr );
168     }
169
170     template< typename Allocator >
171     auto __do_deallocate_unaligned__( Allocator& allocator,
            void* ptr, meta::priority_tag<3> ) -> decltype(
            allocator.deallocate( ptr ) ) {
172       return allocator.deallocate( ptr );
173     }
174
175     template< typename Allocator >
176     auto __do_deallocate_unaligned__( Allocator& allocator,
            void* ptr, meta::priority_tag<4> ) -> decltype(
            allocator_customization::allocator<Allocator>::
            deallocate( allocator, ptr ) ) {
```

```
177         return allocator_customization::allocator<Allocator>::
                deallocate( allocator, ptr );
178       }
179
180       template< typename T >
181       using __try_deallocate_unalign__ = decltype(
              __do_deallocate_unaligned__( std::declval<T&>(), std::
              declval<void*>(), std::declval<meta::priority_tag<4>>()
               ) );
182
183       template< typename T >
184       constexpr bool __can_deallocate_unalign__ = meta::
              is_valid_expression_v<__try_deallocate_unalign__, T>;
185
186       struct __test_deallocate_unalign__ {
187         void dealloc( void* );
188       };
189
190       static_assert( __can_deallocate_unalign__<
              __test_deallocate_unalign__> );
191
192       static_assert( !__can_deallocate_unalign__<int> );
193
194       template< typename Allocator >
195       auto __do_deallocate_size__( Allocator& allocator, void*
              ptr, size_t size, meta::priority_tag<0> ) -> decltype(
              dealloc( allocator, ptr, size ) ) {
196         return dealloc( allocator, ptr, size );
197       }
198
199       template< typename Allocator >
200       auto __do_deallocate_size__( Allocator& allocator, void*
              ptr, size_t size, meta::priority_tag<1> ) -> decltype(
              deallocate( allocator, ptr, size ) ) {
201         return deallocate( allocator, ptr, size );
202       }
203
204       template< typename Allocator >
205       auto __do_deallocate_size__( Allocator& allocator, void*
```

```
        ptr, size_t size, meta::priority_tag<2> ) -> decltype(
        allocator.dealloc( ptr, size ) ) {
206       return allocator.dealloc( ptr, size );
207     }
208
209     template< typename Allocator >
210     auto __do_deallocate_size__( Allocator& allocator, void*
        ptr, size_t size, meta::priority_tag<3> ) -> decltype(
        allocator.deallocate( ptr, size ) ) {
211       return allocator.deallocate( ptr, size );
212     }
213
214     template< typename Allocator >
215     auto __do_deallocate_size__( Allocator& allocator, void*
        ptr, size_t size, meta::priority_tag<4> ) -> decltype(
        allocator_customization::allocator<Allocator>::
        deallocate( allocator, ptr, size ) ) {
216       return allocator_customization::allocator<Allocator>::
          deallocate( allocator, ptr, size );
217     }
218
219     template< typename T >
220     using __try_deallocate_align__ = decltype(
        __do_deallocate_size__( std::declval<T&>(), std::
        declval<void*>(), std::declval<size_t>(), std::declval<
        meta::priority_tag<4>>() ) );
221
222     template< typename T >
223     constexpr bool __can_deallocate_size__ = meta::
        is_valid_expression_v<__try_deallocate_align__, T>;
224
225     struct __test_deallocate_align__ {
226       void dealloc( void*, size_t );
227     };
228
229     static_assert( __can_deallocate_size__<
        __test_deallocate_align__> );
230
231     static_assert( !__can_deallocate_size__<
```

```
    __test_deallocate_unalign__> );
232
233     template< typename T >
234     constexpr bool __can_allocate__ = __can_allocate_align__<T
          > || __can_allocate_unalign__<T>;
235
236     template< typename T >
237     constexpr bool __can_deallocate__ =
          __can_deallocate_size__<T> ||
          __can_deallocate_unalign__<T>;
238
239     static_assert( __can_allocate__<malloc_allocator_t> );
240
241     static_assert( __can_deallocate__<malloc_allocator_t> );
242
243     constexpr auto __allocate__ = []( auto& alloc, size_t sz,
          size_t align ) {
244       using alloc_t = std::decay_t<decltype(alloc)>;
245       if constexpr( __can_allocate_align__<alloc_t> ) {
246         return __do_allocate_aligned__( alloc, sz, align, meta
              ::priority_tag<4>{} );
247       }
248       else {
249         static_assert( __can_allocate_unalign__<alloc_t> );
250         ( void ) align;
251         return __do_allocate_unaligned__( alloc, sz, meta::
              priority_tag<4>{} );
252       }
253     };
254
255     constexpr auto __deallocate__ = []( auto& alloc, void* ptr
          , size_t size ) {
256       using alloc_t = std::decay_t<decltype(alloc)>;
257       if constexpr( __can_deallocate_size__<alloc_t> ) {
258         __do_deallocate_size__( alloc, ptr, size, meta::
              priority_tag<4>{} );
259       }
260       else {
261         static_assert( __can_deallocate_unalign__<alloc_t> );
```

```
262             ( void ) size;
263             __do_deallocate_unaligned__( alloc, ptr, meta::
                    priority_tag<4>{} );
264         }
265     };
266
267 }
268 /**
269 * @endcond
270 *
271 */
272
273 /**
274 * @brief A variable that indicates whenever a type is an
      allocator or not.
275 * @tparam T The type to query.
276 * @details A type is considered an allocator if it can both
      allocate and deallocate memory.
277 * For the memory allocation part, a type must either have a
      member allocate or alloc function or it must have non
      members allocate or alloc functions.
278 * Those functions should allocate either with 2 parameters,
      the size and alignment of the allocation or just the size
       of the allocation.
279 * In case no such functions exist, see geometricks::memory::
      allocator_customization::alocator.
280 *
281 * For the memory deallocation part, a type must either have
      a member deallocate or dealloc function or it must have
      non members deallocate or dealloc functions.
282 * Those functions should deallocate either with 2 parameters
      , the pointer and size of the deallocation or just the
      pointer to deallocate.
283 * In case no such functions exist, see geometricks::memory::
      allocator_customization::alocator.
284 */
285 template< typename T >
286 constexpr bool is_allocator = __detail__::__can_allocate__<T
    > && __detail__::__can_deallocate__<T>;
```

```
287
288     /**
289     * @cond EXCLUDE_DOXYGEN
290     *
291     * Internal not to be documented
292     */
293     namespace __detail__ {
294
295       struct __v_table_for_allocator__ final {
296
297         using __allocate_t__     = void*( void*, size_t, size_t )
                ;
298
299         using __deallocate_t__   = void( void*, void*, size_t );
300
301         __allocate_t__* __allocate__;
302
303         __deallocate_t__* __deallocate__;
304
305       };
306
307       template< typename T >
308       __v_table_for_allocator__* __make_v_table__() noexcept {
309
310         static __v_table_for_allocator__ table = {
311           []( void* ptr, size_t sz, size_t align ) {
312             return __detail__::__allocate__( *static_cast<T*>(
                    ptr ), sz, align );
313           },
314           []( void* ptr, void* obj, size_t size ) {
315             __detail__::__deallocate__( *static_cast<T*>( ptr ),
                    obj, size );
316           }
317         };
318         return &table;
319       }
320
321       static void* __default_allocator__ = &malloc_allocator;
322
```

```
323         static __v_table_for_allocator__* __default_v_table__ =
                __make_v_table__<malloc_allocator_t>();
324
325     }
326     /**
327     * @endcond
328     */
329
330     allocator get_default_allocator();
331
332     /**
333     * @brief Sets the default allocator type.
334     * @tparam T A type that must conform to the allocator
            interface. See @ref is_allocator.
335     * @param alloc The allocator we want to set as the default
            allocator.
336     * @details Sets the default allocator to be used by all
            default constructed geometricks::allocator.
337     * Since all allocators are implemented as views, be extra
            careful not to make the default allocator end its
            lifetime prematurely
338     * or terrible things could happen. This function can also be
             used to temporarely change the default allocator back
            and forth.
339     * Example:
340     * @code{.cpp}
341         struct stack_allocator_t {
342           ...
343         }; //Allocates objects on the stack
344         auto current_default = geometricks::memory::
                get_default_allocator();
345         stack_allocator_t stack_alloc{};
346         geometricks::memory::set_default_allocator( stack_alloc );
              //DANGER! MAKE SURE TO SET THE DEFAULT ALLOCATOR BACK
            OR FACE THE CONSEQUENCES!!!!
347         std::vector<std::tuple<float,int,double>> input_vector;
348         ...
349         geometricks::kd_tree<std::tuple<float,int,double>> tree(
                input_vector.begin(), input_vector.end() ); //KD Tree
```

```
        memory is entirely on the stack now.
350     //Note the same effect could be used just passing the
          stack allocator to the tree constructor in this case.
351     ...
352     geometricks::memory::set_default_allocator(
          current_default ); //WE'RE SAVED!
353   * @endcode
354   * @note The default allocator on startup is set to @ref
        geometricks::memory::malloc_allocator_t "malloc_allocator
        ".
355   */
356   template< typename T >
357   void set_default_allocator( T& alloc );
358
359   void set_default_allocator( allocator& allocator );
360
361 }
362
363 /**
364 * @brief Type erased view to an external allocator.
365 * @details This class doesn't own the actual allocator.
      Instead, it is used as a view to an external allocator and
      stored in the data structures of this project.
366 * This way, the allocator should ALWAYS have at least the same
      object lifetime as the data structure. Not doing so
      results in undefined behavior.
367 * An allocator is defined as anything that can allocate raw
      bytes of memory with either of 2 signatures: it either
      allocates a size of memory or a size of aligned memory.
368 * Example:
369 * @code{.cpp}
370   struct my_aligned_alloc {
371     void* allocate( size_t size, size_t align );
372   };
373   struct my_alloc {
374     void* allocate( size_t size );
375   };
376 * @endcode
377 * In case both functions are supported, it always picks the
```

```
        aligned one.
378     * For that, the type must either supply an allocate function
          or an alloc function or an allocate free function that
          takes a reference to the type as the first parameter
379     * or an alloc free function that takes a reference to the type
           as the first parameter. If none are supplied, see
          geometricks::memory::allocator_customization::allocator.
380     * Optionally, it is also possible the same way to supply a
          reallocate function, for the cases it is prefered to
          reallocate memory.
381     * Note that supplying an allocate function is not enough, as
          the type must also know how to deallocate memory for it to
          work.
382     * For the deallocate functions, you can either supply a
          deallocate function taking a void* or a deallocate function
           taking a void* and a size_t.
383     * Example:
384     * @code{.cpp}
385       struct my_complete_alloc {
386         void* allocate( size_t size );
387         void deallocate( void* );
388       };
389       struct my_complete_aligned_alloc_with_size {
390         void* allocate( size_t size, size_t align );
391         void deallocate( void* ptr, size_t size );
392       };
393     * @endcode
394     * In case both functions are supported, it picks the size one.
395     * The type must either supply a deallocate function or a
          dealloc function. If it doesn't, it can also work with
          deallocate or dealloc free functions taking a reference to
396     * the type as the first parameter. If none are supplied, see
          geometricks::memory::allocator_customization::allocator.
397     * @todo Work on realloc documentation.
398     */
399     struct allocator final {
400
401       /**
402       * @brief Default constructs an allocator view, using the
```

```
        default allocator as the allocator.
403     * @see @ref geometricks::memory::set_default_allocator().
404     * @see @ref geometricks::memory::get_default_allocator().
405     */
406     allocator(): m_allocator( memory::__detail__::
            __default_allocator__ ), m_table( memory::__detail__::
            __default_v_table__ ) {
407     }
408
409     /**
410     * @brief Constructs a view to an allocator.
411     * @tparam Allocator A type that conforms to the allocator
            interface.
412     * @param alloc The allocator itself.
413     * @note This is not the copy constructor.
414     */
415     template< typename Allocator, typename Void = std::
            enable_if_t< !std::is_same_v< std::decay_t<Allocator>,
            allocator > > >
416     allocator( Allocator& alloc ): m_allocator( ( void* ) &alloc
            ), m_table( memory::__detail__::__make_v_table__<
            Allocator>() ) {
417       static_assert( memory::is_allocator<Allocator> );
418     }
419
420     /**
421     * @brief Constructs a view from another view, with both of
            them pointing to the same allocator.
422     */
423     allocator( const allocator& other ): m_allocator( other.
            m_allocator ), m_table( other.m_table ) {
424     }
425
426     friend allocator memory::get_default_allocator();
427
428     friend void memory::set_default_allocator( allocator&
            allocator );
429
430     /**
```

```
431       * @brief Allocates aligned memory.
432       * @param sz The size of the allocation.
433       * @param align The alignment of the allocation.
434       * @returns Aligned raw bytes of memory of size sz.
435       */
436       void* allocate( size_t sz, size_t align = alignof( std::
              max_align_t ) ) {
437         return m_table->__allocate__( m_allocator, sz, align );
438       }
439
440       /**
441       * @brief Deallocates memory.
442       * @param ptr Pointer to memory we want to deallocate.
443       * @param size Size of the allocation. Useful for allocators
              like multipool allocators.
444       */
445       void deallocate( void* ptr, size_t size = 0 ) {
446         m_table->__deallocate__( m_allocator, ptr, size );
447       }
448
449       /**
450       * @brief Compares the view to an allocator.
451       * @tparam Allocator A type that conforms to the allocator
              interface.
452       * @param alloc The allocator.
453       * @returns true if the view points to the allocator. false
              otherwise.
454       */
455       template< typename Allocator >
456       bool operator==( const Allocator& alloc ) const {
457         static_assert( memory::is_allocator<Allocator> );
458         //TODO: pass it to the correct type, maybe?
459         return m_allocator == ( void* )( &alloc );
460       }
461
462       /**
463       * @brief Compares 2 views to allocators.
464       * @param other The other allocator.
465       * @returns true if both views point to the same allocator.
```

```
        false otherwise.
466     */
467     bool operator==( const allocator& other ) const {
468       return m_allocator == other.m_allocator;
469     }
470
471   private:
472
473     allocator( void* ptr, memory::__detail__::
          __v_table_for_allocator__* table ): m_allocator( ptr ),
          m_table( table ) {
474     }
475
476     void* m_allocator;
477
478     memory::__detail__::__v_table_for_allocator__* m_table;
479
480   };
481
482   /**
483   * @brief Free function for allocate.
484   * @see geometricks::memory::allocate( size_t, size_t ).
485   */
486   void* allocate( allocator& alloc, size_t sz, size_t align =
        alignof( std::max_align_t ) ) {
487     return alloc.allocate( sz, align );
488   }
489
490   /**
491   * @brief Free function for deallocate.
492   * @see geometricks::memory::deallocate( void*, size_t ).
493   */
494   void deallocate( allocator& alloc, void* ptr, size_t size = 0
        ) {
495     alloc.deallocate( ptr, size );
496   }
497
498   namespace memory {
499
```

```
500     /**
501     * @brief Returns the default allocator.
502     * @returns A view to the default allocator set by @ref
            set_default_allocator(). If none was set, returns the
            operator new allocator.
503     * @details After a call to @ref set_default_allocator(),
            returns a geometricks::allocator view to the default
            allocator.
504     * @note The default allocator at program startup is always
            set to operator new.
505     * @see geometricks::memory::malloc_allocator_t.
506     */
507     allocator get_default_allocator() {
508       return allocator{ __detail__::__default_allocator__,
              __detail__::__default_v_table__ };
509     }
510
511     template< typename T >
512     void set_default_allocator( T& allocator ) {
513       static_assert( is_allocator<T> );
514       __detail__::__default_allocator__ = ( void* ) &allocator;
515       __detail__::__default_v_table__ = __detail__::
              __make_v_table__<T>();
516     }
517
518     void set_default_allocator( allocator& allocator ) {
519       __detail__::__default_allocator__ = allocator.m_allocator;
520       __detail__::__default_v_table__ = allocator.m_table;
521     }
522
523   }
524
525 }
526
527 #endif //GEOMETRICKS_MEMORY_ALLOCATOR_HPP
```

Código B.16 – geometricks/meta/detect.hpp

```
1  #ifndef GEOMETRICKS_META_DETECT_HPP
2  #define GEOMETRICKS_META_DETECT_HPP
3
4  //C++ stdlib includes
5  #include <type_traits>
6
7  namespace geometricks {
8
9    namespace meta {
10
11     namespace __detail__ {
12
13       template< template< typename... > typename Expression,
             typename AlwaysVoid = void, typename... Types >
14       struct __is_valid_expression_impl__ : std::false_type {};
15
16       template< template< typename... > typename Expression,
             typename... Types >
17       struct __is_valid_expression_impl__< Expression, std::
             void_t< Expression< Types... > >, Types... > : std::
             true_type { };
18
19       template< typename T, typename... Params >
20       using __counstruct_with__ = decltype( T( std::declval<
             Params>()... ) );
21
22     }
23
24
25     template< template< typename... > typename Expression,
         typename... Types >
26     using is_valid_expression = __detail__::
           __is_valid_expression_impl__< Expression, void, Types...
           >;
27
28     template< template< typename... > typename Expression,
         typename... Types >
29     constexpr bool is_valid_expression_v = is_valid_expression<
```

```
        Expression, Types... >::value;
30
31      template< typename T, typename... Params >
32      using is_constructible_with = is_valid_expression<
            __detail__::__counstruct_with__, T, Params... >;
33
34      template< typename T, typename... Params >
35      constexpr bool is_constructible_with_v =
            is_constructible_with< T, Params... >::value;
36
37    }
38
39 }
40
41 #endif //GEOMETRICKS_META_DETECT_HPP
```

Código B.17 – geometricks/meta/utils.hpp

```
1  #ifndef GEOMETRICKS_META_UTILS_HPP
2  #define GEOMETRICKS_META_UTILS_HPP
3
4  #include "detect.hpp"
5
6  namespace geometricks {
7
8    struct default_compare_t {} default_compare;
9
10   namespace meta {
11
12     //Helper metafunction that wraps a result in a named type
         type.
13     //Using this metafunction is preferred over typing manually
         to avoid potential typing errors.
14     template< typename T >
15     struct result_is {
16
17       using type = T;
18
19     };
20
21     namespace __detail__ {
22
23       template< typename T >
24       struct __print_impl__ {
25         using ReceivedType = typename T::
              __FORCECOMPILATIONERROR__;
26       };
27
28     }
29
30     //Function to print a type at compile time
31     //We make it raise an error to abort compilation and get
         useful information.
32     template< typename T >
33     using print = typename __detail__::__print_impl__< T >::
         ReceivedType;
```

```
34
35      template< typename T >
36      constexpr bool always_false = false;
37
38      template< typename T >
39      struct numeric_limits : std::numeric_limits< T > {};
40
41      template< int I >
42      struct priority_tag : public priority_tag<I - 1> {};
43
44      template<>
45      struct priority_tag<0> {};
46
47      template< typename T >
48      struct insert_tag;
49
50      struct push_back {};
51
52      struct insert {};
53
54      template< typename T >
55      using push_back_expr = decltype( std::declval<T>().push_back
           ( std::declval<typename T::value_type>() ) );
56
57      template< typename T >
58      using insert_expr = decltype( std::declval<T>().insert( std
           ::declval<typename T::value_type>() ) );
59
60      template< typename T >
61      constexpr bool has_push_back = is_valid_expression_v<
           push_back_expr, T>;
62
63      template< typename T >
64      constexpr bool has_insert = is_valid_expression_v<
           insert_expr, T>;
65
66      static_assert( has_push_back<std::vector<int>> );
67
68      template< typename T,
```

```
69                typename Container >
70      void add_element( T&& element, Container& cont ) {
71        if constexpr( has_push_back<Container> ) {
72          cont.push_back( std::forward<T>( element ) );
73        }
74        else if constexpr( has_insert<Container> ) {
75          cont.insert( std::forward<T>( element ) );
76        }
77        else {
78          static_assert( geometricks::meta::always_false<T> );
79        }
80      }
81
82    }
83
84 }
85
86 #endif //GEOMETRICKS_META_UTILS_HPP
```