A data structure for managing OpenGL vertex buffers

Michał Szopiński, BSc

March 24, 2024

Abstract

This document is a write-up of an algorithmic problem that I encountered while working on a hobby project of mine. Described herein is a data structure for managing contiguous memory for the purpose of drawing vertices in OpenGL.

1 Overview

To keep myself busy while unemployed, I've been working on a video game with a voxel-based world similar to Minecraft. The game uses OpenGL to render the world cube-by-cube. Arbitrary modifications to the world are allowed, which means that cubes may appear, disappear or be modified at any position at any moment.

OpenGL exposes the `drawArrays` function as the primary means of drawing vertices. In great simplicifcation, the function accepts a pointer to a contiguous buffer of arbitrary data and applies a user-defined algorithm to render primitives based on that data. In this case, a single entry in the buffer describes a single cube.

When the world is first loaded, the buffer is filled with thousands of cubes. When a cube is removed from the world, the array may be shortened, or a "gap" between elements may appear. Entries inside the gap are adjusted to instruct OpenGL not to draw any vertices based on those entries.

The game performs best when the least amount of cubes are drawn. Care must be taken not to expand the buffer, and when new cubes are added, any existing gaps must be reused if possible. This creates the need for a data structure that would make it possible to quickly find unallocated gaps in a contiguous buffer.

The solution I designed uses a tree data structure to keep track of which parts of an address space of buffer indices are currently in use (fig. 1). Each node uses a 64-bit integer to keep track of 64 divisions of the address space. Terminal (leaf) nodes keep track of individual addresses. A high n-th bit indicates that the n-th child node is fully in use, a low bit indicates that the child node may be used for further allocations.

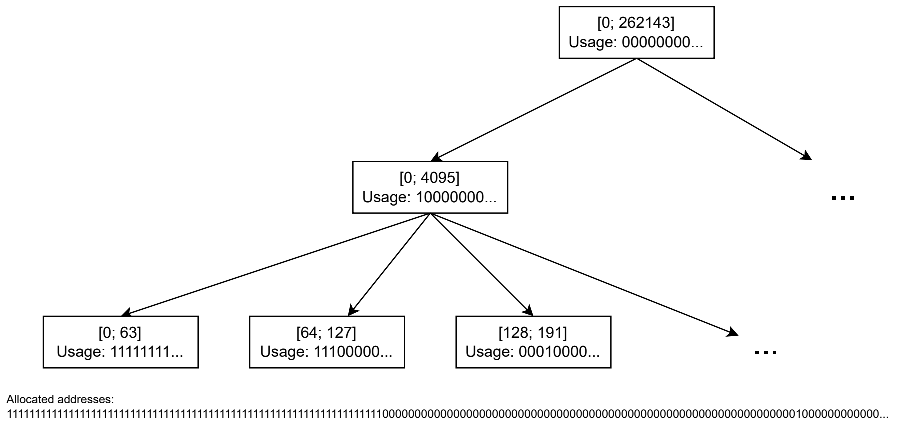

Figure 1: Tree data structure for keeping track of allocated addresses

Efficient allocation is possible thanks to `std::countr_one`, which on x86 architectures is implemented using a single CPU instruction. Allocation begins by asking the root node to allocate an address. The root node finds the first unused child using `std::countr_one` and asks that child to perform the allocation. Eventually, a terminal node is reached, which marks a single address as allocated and updates usage data in parent nodes.

Deallocation is implemented in a similar manner. The root node is asked to deallocate a specific address. The node calculates which child keeps track of the specified address and delegates the deallocation to that child. Eventually, a terminal node is reached, which updates its own and its parents' usage data.

The tree structure is hidden behind a wrapper class named `kh::AddressSpace`. The class implements an optimization that keeps track of which terminal node was previously used to allocate an address, which speeds up bulk allocation.

When the address space is fully used and an attempt at allocation fails, the wrapper class calls `escalate` on the root node. This causes the root node to consolidate all of its children into a single child and expands the root node's range. The structure can therefore handle infinite address spaces.

To limit the amount of memory consumed by a single node, each non-terminal node allocates an array of 64 sub-nodes and constructs them as necessary. By doing so, it only needs one pointer to maintain a reference to all of its children. Because the nodes are numerous and subject to frequent allocation, the memory for their storage is allocated by a `std::pmr::polymorphic_allocator` combined with a `std::pmr::unsynchronized_pool_resource`.

2 Code

2.1 address_space.hpp

```cpp
#ifndef ADDRESS_SPACE_HPP
#define ADDRESS_SPACE_HPP

#include <bit>
#include <cstdint>

namespace kh {
    /*
        @brief Class for quickly assigning and unassigning addresses in a contiguous range.
    */
    class AddressSpace {
        public:
            /*
                @brief Assign an address in the address space.

                @returns The assigned address
            */
            std::size_t allocate();

            /*
                @brief Free an address in the address space.

                @param address The address to be deallocated
            */
            inline void deallocate(std::size_t address) {
                root.deallocate(address);
            }

        private:
            class AllocationNode {
                public:
                    /*
                        @brief Create an allocation node.

                        @param rangeStart Lowest address managed by this node
                        @param parent Parent node
                        @param divisor Address space division level
                    */
                    AllocationNode(std::size_t rangeStart = 0, AllocationNode *parent = nullptr, int divisor =
                         0);

                    AllocationNode(AllocationNode&) = delete;
                    AllocationNode& operator=(AllocationNode&) = delete;

                    AllocationNode(AllocationNode&&);
                    AllocationNode& operator=(AllocationNode&&);

                    ~AllocationNode();

                    /*
                        @brief Allocate an address in this node or in any of its children.

                        @param[out] The terminal node used to peform the allocation, or nullptr if the
                             allocation failed
                        @returns The assigned address. If the allocation failed, the value is undefined.
                    */
                    std::size_t allocate(kh::AddressSpace::AllocationNode *&terminalNode);

                    /*
                        @brief Deallocate an address in this node or in any of its children.

                        @param address The address to be deallocated
                    */
                    void deallocate(std::size_t address);

                    /*
                        @brief Expand the tree by creating a new node and moving all children into it.
                    */
                    void escalate();

                private:
                    /*
                        @brief Mark the specified child as used. If the node becomes fully allocated, update
                             the parent.
                    */
                    void markChildUsed(AllocationNode* child);

                    /*
                        @brief Mark the specified child as unused. If the node becomes partially unallocated,
                             update the parent.
                    */
                    void markChildUnused(AllocationNode* child);

                    /*
                        @brief Check if the node can be used to allocate any more children.
                    */
                    inline bool isFullyAllocated() const {
                        return std::countr_one(usage) == 64;
                    }

                    std::size_t rangeStart;
                    int divisor;

                    AllocationNode *parent = nullptr;
                    AllocationNode *children;
                    int childCount = 0;

                    uint64_t usage = 0;
            };

            /*
                @brief The top-level allocation node.
            */
            AllocationNode root;

            /*
                @brief Reference to the last node used for allocation. Optimizes bulk allocation.
            */
            AllocationNode *lastTerminal = nullptr;
    };
}

#endif
```

2.2 address_space.cpp

```
#include "address_space.hpp"
#include "memory.hpp"

std::size_t kh::AddressSpace::allocate() {
    // attempt to reuse last allocation node
    if (lastTerminal != nullptr) {
        std::size_t address = lastTerminal->allocate(lastTerminal);

        if (lastTerminal != nullptr)
            return address;
    }

    // if the last allocation node is unavailable, use the root node
    std::size_t address = root.allocate(lastTerminal);

    if (lastTerminal != nullptr)
        return address;

    // if root node allocation failed, expand tree and try again
    root.escalate();
    return root.allocate(lastTerminal);
}

kh::AddressSpace::AllocationNode::AllocationNode(std::size_t rangeStart, AllocationNode *parent, int divisor)
    : rangeStart(rangeStart), parent(parent), divisor(divisor) {
    // allocate memory for 64 children if the node is not terminal
    if (divisor == 0)
        return;

    children = std::pmr::polymorphic_allocator<AllocationNode>(&allocationNodePool).allocate(64);
}

kh::AddressSpace::AllocationNode::AllocationNode(AllocationNode&& other) {
    *this = std::move(other);
```

```
}

kh::AddressSpace::AllocationNode &kh::AddressSpace::AllocationNode::operator=(AllocationNode&& other) {
    rangeStart = other.rangeStart;
    divisor = other.divisor;
    parent = other.parent;
    children = other.children;
    other.children = nullptr;
    childCount = other.childCount;
    other.childCount = 0;
    usage = other.usage;

    return *this;
}

kh::AddressSpace::AllocationNode::~AllocationNode() {
    // destroy each constructed child
    for (int i = 0; i < childCount; i++)
        std::destroy_at(children + i);

    // deallocate memory for children
    if (children != nullptr)
        std::pmr::polymorphic_allocator<AllocationNode>(&allocationNodePool).deallocate(children, 64);
}

std::size_t kh::AddressSpace::AllocationNode::allocate(AllocationNode *&terminalNode) {
    // check if allocation possible
    if (isFullyAllocated()) {
        terminalNode = nullptr;
        return 0;
    }

    // find unallocated index
    int freeIndex = std::countr_one(usage);

    // if this is a terminal node, allocate an address
    if (divisor == 0) {
        usage |= 1ULL << freeIndex;

        // update parent usage
        if (isFullyAllocated() && parent != nullptr)
            parent->markChildUsed(this);

        terminalNode = this;
        return rangeStart + freeIndex;
    }

    // if not a terminal node, ask a child to allocate an address
    kh::AddressSpace::AllocationNode *child = children + freeIndex;

    // ensure that the child is constructed
    if (freeIndex >= childCount) {
        std::size_t childRangeStart = rangeStart + (static_cast<std::size_t>(freeIndex) << divisor);

        std::construct_at(child, childRangeStart, this, divisor - 6);
        childCount++;
    }

    // delegate allocation to the child
    std::size_t output = child->allocate(terminalNode);

    // update parent usage
    if (isFullyAllocated() && parent != nullptr)
        parent->markChildUsed(this);

    return output;
}

void kh::AddressSpace::AllocationNode::escalate() {
    // create a new node to replace this one
    AllocationNode newThis(rangeStart, nullptr, divisor + 6);
    newThis.childCount = 1;
    newThis.usage = isFullyAllocated();
    newThis.children[0] = std::move(*this);
    newThis.children[0].parent = this;

    // update the parent pointer on grandchild nodes
    for (int i = 0; i < newThis.children[0].childCount; i++)
        newThis.children[0].children[i].parent = &newThis.children[0];

    *this = std::move(newThis);
```

```
}

void kh::AddressSpace::AllocationNode::markChildUsed(AllocationNode* child) {
    // calculate child index and mark it as used
    usage |= 1ULL << (child - children);

    // update parent usage
    if (isFullyAllocated() && parent != nullptr)
        parent->markChildUsed(this);
}

void kh::AddressSpace::AllocationNode::deallocate(std::size_t address) {
    // find out which child or bit is affected
    int index = (address - rangeStart) >> divisor;

    // notify the parent that this child is no longer fully used
    if (isFullyAllocated() && parent != nullptr)
        parent->markChildUnused(this);

    // update usage
    if (divisor == 0)
        usage &= ~(1ULL << index);
    else if (index < childCount)
        children[index].deallocate(address);
}

void kh::AddressSpace::AllocationNode::markChildUnused(AllocationNode* child) {
    // update parent usage
    if (isFullyAllocated() && parent != nullptr)
        parent->markChildUnused(this);

    // calculate child index and mark it as used
    usage &= ~(1ULL << (child - children));
}
```

3 Tests

To test the data structure, the following code was run:

```
AddressSpace space;

for (int i = 0; i < 190; i++)
    std::cout << "allocated␣" << space.allocate() << std::endl;

space.deallocate(13);
space.deallocate(17);
space.deallocate(70);
space.deallocate(71);
space.deallocate(150);
space.deallocate(152);
std::cout << "deallocated␣13,␣17,␣70,␣71,␣150,␣152" << std::endl;

for (int i = 0; i < 10; i++)
    std::cout << "allocated␣" << space.allocate() << std::endl;
```

The output was as follows:

```
...
allocated 185
allocated 186
allocated 187
allocated 188
allocated 189
deallocated 13, 17, 70, 71, 150, 152
allocated 150
allocated 152
allocated 190
allocated 191
allocated 13
allocated 17
allocated 70
allocated 71
allocated 192
allocated 193
```

Following the deallocation, the first two allocations (150, 152) could be attributed to the “last terminal” optimization, which filled out the gaps in the previously used terminal node. The two subsequent allocations (190, 191) used up the remaining two places in the final terminal node.

Once the previously used terminal node was exhausted, allocations proceeded from the root node. Gaps 13, 17, 70 and 71 were filled. Eventually, there were no more gaps, and allocations proceeded from a new terminal node $[192; 255]$.

The data structure was successfully used to implement cube rendering (fig. 2).

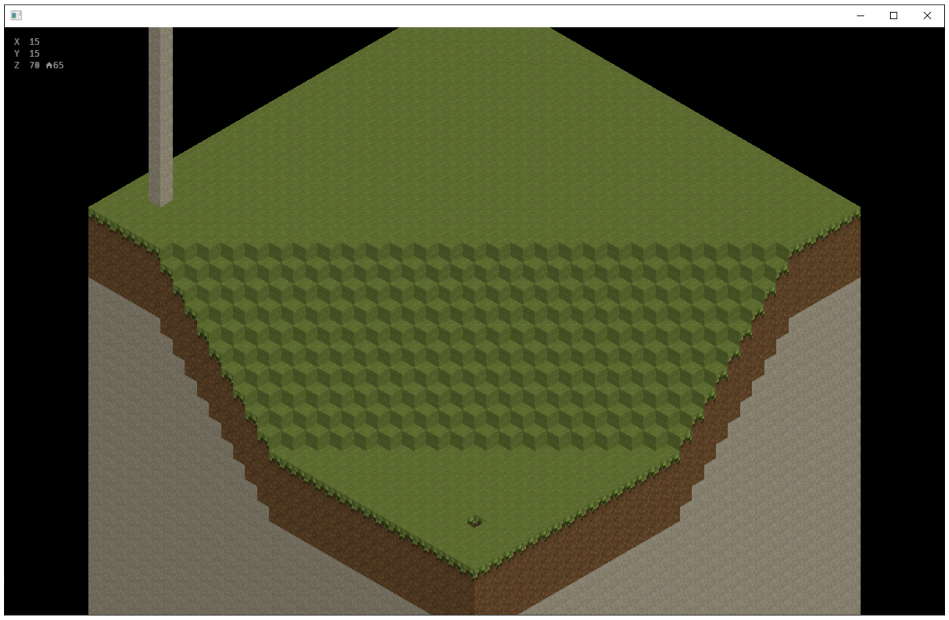

Figure 2: A screenshot of the game.