Fall 2015

An Investigation in Implementing a C++ Voxel Game Engine with Destructible Terrain

Michael W. Wilder
University of Akron Main Campus, mww12@zips.uakron.edu

An Investigation in Implementing a C++ Voxel Game Engine with Destructible Terrain

Michael Wilder
The University of Akron

Abstract

In video games today, world geometry is often represented as a polygon mesh. While this representation is able to represent terrain, it must be done so using an elevation grid. For each X and Z location, an elevation grid stores the height of the terrain at that location. This has a major drawback: it is unable to allow terrain with overhangs and caves, and is unable to allow for destructible terrain. In this project, an alternative technique using voxels is explored to overcome these limitations for a game engine. A voxel is the 3D equivalent of a pixel and allows for 3D model manipulation by adding and removing discrete units. Because voxel objects require O(n^3) memory to represent, a number of challenges must be overcome to allow efficient manipulation, including effective use of cache memory, minimization of polygon count, efficient communication between the CPU and GPU, efficient collision detection when brute force methods require O(n^6) collision checks, and deletion of triangles from fixed-size Vertex Buffer Objects. The purpose of this project is to investigate and create solutions for these problems and successfully implement a simple game engine in C++ that allows for destructible terrain and terrain generation from mathematical formulas.

Keywords: voxel, game engine, 3D, destructible terrain, collision detection, terrain generation

Introduction

The word “pixel” is a combination of the words “picture” and “element”, meaning the smallest individual element in a 2D picture[1]. In computer graphics, a pixel is represented by the color value at each location in a 2D grid. In an RGB color space, this is represented as the intensity of the primary colors red, green, and blue light mixed to create the desired color. It should be noted that in almost all cases, computer monitors are flat; thus, all rasterization, even that which appears 3D, inherently happens in 2D space. To go from 3D geometry to a computer monitor, the geometry must be projected onto a 2D screen and rasterized as triangles[2].

However, what if we extend the idea of a pixel to a higher dimension? For the purposes of 3D graphics, instead of having images composed of small rectangular “picture elements”, we have volumes constructed of 3D regular hexahedrons known as “volume elements”, or voxels[1]. Each voxel represents

the value at that location, which could could e.g. a color, a density from a CT scan, or a "block" as in the game Minecraft. This allows things such as destructible terrain, dynamic terrain generation from mathematical formulas, collision detection for irregularly shaped objects, and the creation of complex map geometry that has terrain features with e.g. overhangs and caves.

Purpose

Since I first began programming, I have always had a strong interest in computer graphics and game development. In recent years, my interests have shift towards 3D rendering and the creation of game engines. Typically, geometry in a game is represented as a collection of polygons in what are known as *polygon meshes*[2]. In a voxel engine, however, it is managed as a 3D grid of discrete cubes. Though it has been used in a few games, such as Minecraft, voxels have not had much use in video games. Because of this, there are several opportunities research into new methods. When I first heard of voxel rendering, I found this approach to be very unique and intriguing; I knew at once that I wanted to study it in greater detail. Thus, this project was chosen to further my knowledge in several areas, including voxel rendering and data management, destructible terrain, game engine management, OpenGL, and 3D math. It is hoped that this project can form the basis for future graduate study.

Choice of Development Environment

For the game engine, C++ was the language of choice for a number of reasons. First, the language is low-level enough to allow efficient and direct manipulation of resources, but high-level enough to still allow for things such as abstraction, encapsulation, polymorphism, and templates. As the complexity of a game engine can be quite high with many "moving parts", it is very important to have a modular and flexible design. Second, C++ follows the ideology that only language features which have been requested cause an introduction of overhead. For example, a virtual method table is only generated for a class if some of its methods have been explicitly declared "virtual" by the programmer[3]. This means the programmer is open to choosing the best implementation based on the requirements without being penalized for unneeded features.

One of the end goals of this project was to have a cross-platform game engine that could compile and

run on any platform with minimal effort. To support this goal, the engine uses OpenGL as the backend for 3D rendering, as it is cross-platform, robust, free, and highly efficient with working with large quantities of polygons[4]. In addition, to facilitate platform-independent building, the project uses CMake instead of a hand-written Makefile. CMake is an open-source build system manager that is compiler and operating system independent[5]. Thus, it is perfectly suited for the project.

Efficient Traversal of Large Data Sets

At its core, a 3D grid is a one-dimensional array with the three-dimensional index mapped into this one-dimensional space[7]. This mapping is possible because the grid is a uniform rectangular prism. To understand this mapping, it is simpler to look at the 2D case first i.e. mapping a 2D index into a 1D array. First, consider this 1D array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Notice how each data value corresponds to its index in the array. Now, suppose the array were to be rearranged into a 2D array such that it has four rows and four columns:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Since the 2D array has a uniform number of rows and columns, we can determine the 1D index from the 2D index using the following formula:

$$i = r \cdot c_{total} + c$$

Where r is the row, c is the column, and c_{total} is the number of columns. Suppose that we wanted to visit every element in the two-dimensional array and print it out. There are two similar ways to do this, both involving the use of a double for-loop. The first is to have the outer for-loop variable represent the column and the inner for-loop variable represent the row. A piece of code implementing this method might look like this:

```
int data[16] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
for(int c = 0; c < 4; ++c)
      for(int r = 0; r < 4; ++r)
            cout << data[r * 4 + c] << " ";
```

Running this code would produce the following output:

```
0 4 8 12 1 5 9 13 2 6 10 14 3 7 11 15
```

It is easy to see that this approach leads to the index "jumping around" i.e. there is no consistent linear progression between one index and the next. This is significant for two reasons. First, many Instruction Set Architectures (ISA's) have an indexing mode which allows the address in a register to be incremented after a read or write instruction e.g. the Motorola 68k ISA[6]. If there is a consistent linear progression between array index accesses, the compiler can use this indexing mode; because it consumes less CPU cycles, it leads to greater efficiency[6]. However, with this traversal method, the compiler is forced to do one of two things: either evaluate the formula directly, which involves a multiplication and an addition, or have more complex logic to evaluate the index[7].

Even more significant, however, is the second point: this method introduces the possibility of causing frequent cache misses[8]. *Cache memory* is special high-speed RAM that attempts to solve the conflicting goals of having a large quantity of cheap memory and having memory that is fast[8]. It does this by exploiting what is known as the *Principe of Locality*, which states that if a memory location has recently been accessed, neighboring memory locations are likely to be subsequently accessed[8]. Thus, when a memory location has been accessed from regular RAM, the data at that location and its neighboring data (known as a *cache block*) is copied into the cache memory; this allows subsequent accesses to be served by the faster cache memory instead of the slower regular RAM[8].

If a memory location that is to be accessed is currently in cache, this is called a *cache hit*[8]. If, on the other hand, the data is currently in RAM, this is known as a *cache miss*[8]. A cache miss is significant because the program must be stalled while the entire cache block is copied into cache[8]. Unfortunately, cache memory is also very constrained; there is only room for a limited number of cache blocks in the

cache memory. If a block needs to be loaded into cache and there are no blocks available, the cache controller must choose a block to be overwritten, write the data currently there back to RAM, then load in the new block[8]. Thus, for efficiency, a programmer must effectively utilize memory by minimizing the number of cache misses and writing programs in such a way that they have good locality. Because the first method jumps around so much, each index could very well be located in a different cache block.

There is, however a better approach: simply have the outer for-loop variable be the row and the inner for-loop be the column:

```
int data[16] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
for(int r = 0; r < 4; ++r)
      for(int c = 0; c < 4; ++c)
            cout << data[r * 4 + c] << " ";
```

When run, this code produces the following output:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

As can be seen here, this second approach exhibits good locality and a clear linear progression in index. In 3D, this becomes even more important because instead of two index variables, there are three. This leads to the following formula:

$$i = d \cdot r_{total} \cdot c_{total} + r \cdot c_{total} + c$$

Where d is the depth value, r is the row, and c is the column. For an $n \times n \times n$ grid, the memory requirements are on the order of $O(n^3)$. The larger the dimensions, the more potential there are for cache misses if the data is not traversed in a locality-conscious manner. Thus, throughout the project, an important emphasis was put on traversing the data correctly, using the following approach:

```
for(int d = 0; d < 4; ++d)
```

```
    for(int r = 0; r < 4; ++r)
        for(int c = 0; c < 4; ++c)
            cout << data[d * 4 * 4 + r * 4 + c] << " ";
```

Triangulation of a Voxel Grid

Graphics cards have been optimized to work with triangles, as they are relatively simple to rasterize and have the attractive property that they are always convex[9]. When working with any representation of 3D data, it is necessary to "triangulate" the data i.e. convert the data into a 3D mesh composed of triangles for rasterization[9]. Since the game engine uses voxels that are uniformly-sized cubes, this process is quite simple. Each face of the cube is a quadrilateral, which is decomposed into two triangles. Since a cube has six faces, this process is repeated 6 times.

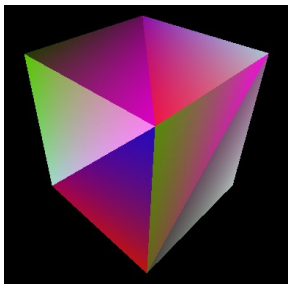

Figure 1: A voxel decomposed intro triangles

When using a single voxel, every triangle in the grid needs to be generated. However, if there are multiple voxels, triangles do not need to be generated for the face between adjacent voxels.

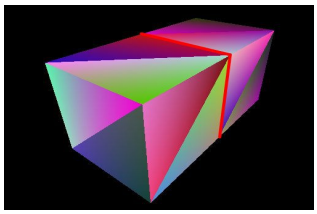

Figure 2: No triangles need to be generated for the face between two adjacent voxels

Thus, to determine whether the triangles should be generated for the given face of a voxel, it is sufficient to check whether or not there is a voxel attached to that face: if there is, there is no need to generate the triangles. As the volume is a 3D grid, this is quite simple: for voxel at position (x, y, z) in the grid, check the following locations to determine whether the triangles should be generated for that

face:

$$\text{top} \rightarrow (x, y - 1, z)$$
$$\text{bottom} \rightarrow (x, y + 1, z)$$
$$\text{left} \rightarrow (x - 1, y, z)$$
$$\text{right} \rightarrow (x + 1, y, z)$$
$$\text{back} \rightarrow (x, y, z + 1)$$
$$\text{front} \rightarrow (x, y, z - 1)$$

This algorithm results in a substantial reduction in the number of triangles that are generated. For a 100 x 100 x 100 grid, assuming there were no empty cells, this would require 12,000,000 triangles to be generated, at a cost of 108 MB of RAM just to represent the vertex data. However, using this method, only 120,000 thousand triangles need be generated.

Once the triangles have been generated, they need to be uploaded to the graphics card using OpenGL[10]. In previous versions of OpenGL, triangles to be rendered were passed to OpenGL for every frame; this led to the CPU wasting much of its time communicating with the graphics card[10]. In modern versions of OpenGL, however, the graphics card can actually be programmed in a language called GLSL (OpenGL Shading Language)[10]. This means that the triangle data can be entirely stored and processed on the graphics card in what is known as a *Vertex Buffer Object* (VBO)[10]. As a result, the CPU can spend its time doing other things.

Generation of Primitives

A *primitive* is a basic 3D shape, such as a sphere, a cube, or a cone[11]. The generation of primitive shapes is a very important part of a game engine, for a number of reasons. First, during development, it provides a simple way to test the features currently in the engine. For example, when the triangulation algorithm was under development, a voxel sphere was used because of its simplicity to generate and the relative complexity of the voxelized volume. Second, primitives can be used to construct more complex objects. A simple example is a snowman, which can be created out of three spheres for the body, two spheres for the eyes, and a cone for the nose.

The easiest primitive to generate is, unsurprisingly, the rectangular prism. Simply create a uniform grid and set each element to not be empty. The last parameter of the Grid3D class is the default value to initialize the grid with, so we simply set it to a value to represent “not empty”. In this case, we use 1:

```
Grid3D<int> g(30, 30, 30, 1, 1, 1, 1);
```

Here, we create a 3D integer grid that has size 30 x 30 x 30, has voxels that are 1 x 1 x 1 unit, and has every voxel initialized to a value of 1. Next, we call the grid's triangulate method, specifying what value should be considered empty:

```
g.triangulate(0);
```

The results of the triangulation can be seen here:

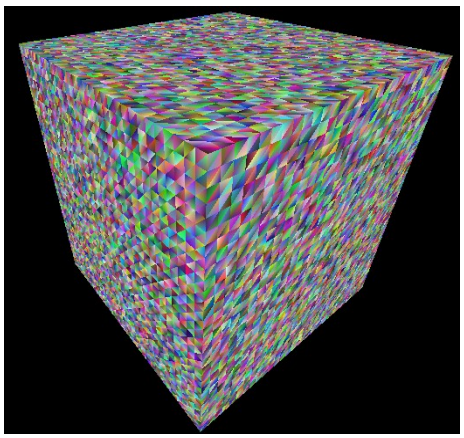

Figure 3: Triangulation of a rectangular prism

The generation of spheres, on the other hand, is much more interesting. To make the generation of primitives more simple, a generation method was created that visits every voxel in the grid and calls a callback function (*eval*) which evaluates to the value of that voxel. The called function is passed to the generate method via the function pointer *eval*.

```
template<typename T>
void Grid3D<T>::generate(T (*eval)(int x, int y, int z, Grid3D<T>& g)) {
```

```
    T* ptr = data;

    for(int z = 0; z < z_size; ++z) {
      for(int y = 0; y < y_size; ++y) {
        for(int x = 0; x < x_size; ++x) {
          *ptr = eval(x, y, z, *this);
          ++ptr;
        }
      }
    }
  }
```

Now, to generate a sphere, we use the definition of a solid sphere i.e. every point that is inside or on the sphere has a distance to the center point that is less than or equal to the radius of the sphere. The sphere generation code looks as follows:

```
template<typename T>
static T generateSphere(int x, int y, int z, Grid3D<T> &g) {
    int r = std::min(g.x_size, std::min(g.y_size, g.z_size)) / 2;

    x -= r;
    y -= r;
    z -= r;

    return x * x + y * y + z * z <= r * r;
  }
```

The first step is to to calculate the radius of the sphere. While it may seem unnecessary to calculate this for each voxel, a voxel grid is not intrinsically a sphere; thus, this radius information is not stored in the grid. As for calculating the radius, it is desired to have the entire sphere fit inside the grid structure. Thus, we calculate the radius of the maximum sphere that will fit inside the grid by finding the minimum dimension and halving it; this works because the perpendicular distance between the center point and the planes that compose the sides of the box is the dot product between the plane normal and the center point plus the distance from the plane to the origin. As these planes are axis-aligned, we can assume that half the maximum dimension is the radius of the sphere.

Next, we need to move the point so that it is relative to the center of the sphere. From here, we check if the distance from the point is less than or equal to the radius. However, the distance formula involves calculating the square root, a notoriously slow operation. To get around this, we instead check if the distance squared is less than the radius squared. Since distance is always positive, we are justified in doing this optimization.

To generate a sphere that has a radius of 25 voxels, we create a grid that is 50 x 50 x 50 voxels. Then, we pass the generateSphere callback into the Grid3D::generate method and finally triangulate the result:

```
Grid3D<int> g(50, 50, 50, 1, 1, 1 , 1);
g.generate(Grid3D_Helper<int>::generateSphere);
g.triangulate(0);
```

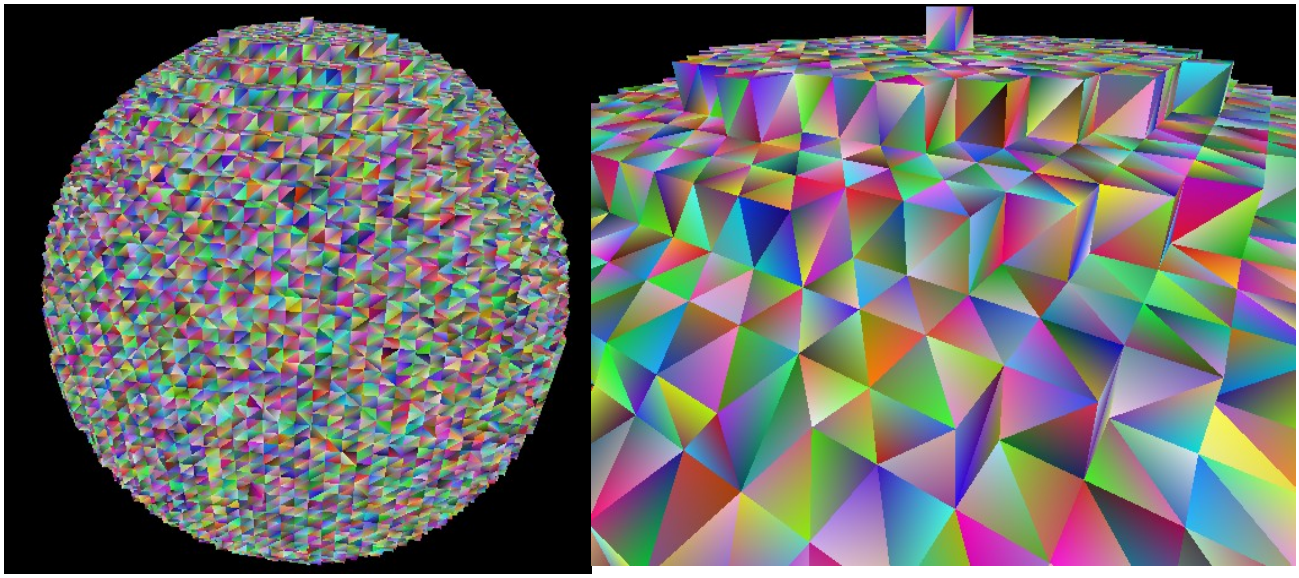

Figure 4: A voxel sphere (left) and closeup (right)

Shape Generation Using Mathematical Formulas

After a time it was realized that hard-coding primitive formulas for generating shapes was rather

inflexible. This is because it requires recompiling the program and writing new code every time the user wants to create a new shape. To solve this, a simple stack-based expression evaluator was implemented that allows the user to enter formulas and have it generate shapes based on it. For simplicity and speed of evaluation, the expression parser only accepts formulas that are in postfix notation instead of infix. This means that instead of entering:

5 + 7

The user would have to input:
5 7 +

The reason that postfix notation was chosen is that it is very easy and efficient to evaluate using a stack[12]. Even though it is more difficult for a user to enter a formula in this notation, the speed increase is very important: for a 100 x 100 x 100 voxel object, this formula will be evaluated 1,000,000 times.

To evaluate a post-fix formula using a stack, the algorithm works as follows[12]:

```
for each token:
      if the token is a number or named constant:
            push it onto the stack
      if the token is a variable:
            evaluate the value of that variable and push it onto the stack
      if the token is a function:
            for as many parameters as the function takes:
                  pop values off the stack into parameter variables
            evaluate the function
            push the return value onto the stack
      if the function is an operator:
            if the operator is a binary operator:
                  pop two values off the stack
                  apply the operator
                  push the result back onto the stack
            else:
                  pop one value off the stack
```

```
                    apply the operator (e.g. 'not')
                    push the result back onto the stack

The value on the top of the stack is the result of evaluating the expression
```

The evaluator supports basic arithmetic operators (addition, subtraction, multiplication, division, power, square root), logical operators (and, or, not), comparison (equal, less than, greater than, less than or equal to, greater than or equal to), and trig functions (sin, cos, tan). In addition, the evaluator automatically defines the numerical constants PI and E. As for variables, it defines the variables

x, y, z → the current index in the 3D voxel grid
cx, cy, cz → the position in the grid relative to the center of the grid
r → the radius of the largest sphere the voxel grid could hold
sr → the distance from the center of the grid to the current voxel ("sphere radius")
cr → the radius a 2D circle would have in the XZ plane if it went through the current voxel

For boolean operators, such as the equality operator, they evaluate to 0 if the condition is false and 1 if the condition is true. This allows conformance with C++ operator semantics. In addition, it allows for every value on the expression stack to be represented as a floating point number.

To create a sphere, a formula like the following could be used:

`x x * y y * z z * + + sqrt r <` → $\sqrt{x^2+y^2+z^2}<r$

Of course, it is also possible to generate more interesting shapes. For example, to generate terrain that has hills, a formula like this can be used:

`cx 5 / sin 5 * cz 5 / sin 5 * + cy =` → $y=5\sin(\frac{x}{5})+5\sin(\frac{z}{5})$

The results of which look like this:

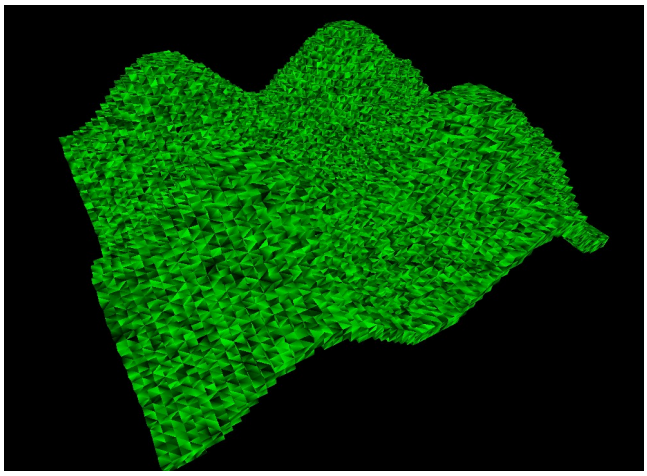

Figure 5: Terrain with hills generated from a postfix equation

Collision Detection

One of the goals for the engine was not only to detect when a collision has occurred between two voxel objects, but to also detect which individual voxels had collided. This is useful for things such as tunneling through terrain by deleting overlapping voxels. This problem, however, is much more complex than it may seem. Collision detection of individual objects is inherently a $O(n^2)$ algorithm; every object has to be checked for a collision between every other object[13]. This makes the brute-force method of collision detection infeasible. To see why, consider two voxel cubes that are each 64 x 64 x 64 voxels. Each cube has 262,144 voxels. To prevent duplicate collision checks, you need exactly n(n + 1) / 2 checks. In total, this would amount to doing 34,359,869,440 checks.

In addition to the problem of the number of checks, there is also the problem of how to determine if two voxels intersect. If we have two axis-aligned cubes (each face of the cube is aligned to the coordinate axes), this is very simple: it is sufficient to check if the bounds of each cube overlap. If, however, we have two cubes that are in different orientations, it is not efficient to check whether they intersect. One possible solution to this to put an Axis-Aligned Bounding Box (AABB) around each voxel, which is the minimum-sized axis-aligned rectangular prism that contains all the points of the voxel[13]. Though this adds the desired property of simple intersection checks, it does introduce another problem. When rotating an object, its AABB can change; this means it must be recalculated every time the object is rotated[13].

To solve the problem of voxel-voxel intersection, the best choice was to use a bounding sphere around each voxel. A *bounding sphere* is the sphere with the minimum radius that is big enough to contain all the vertices of the object it is bounding[14]. Bounding spheres have two properties that make them a very efficient choice. First, a bounding sphere does not change its radius or position when the object it bounds is rotated around its center point[14]. Second, the intersection test is relatively simple: two spheres intersect if the distance between their center points is less than or equal to the sum of their radii[14]. If calculated in the conventional way, this would require a square root calculation. As with the generation of a voxel sphere, though, we can check if the distance squared is less than or equal to the sum of the radii squared. This is valid to do because distance is always positive.

It is still necessary to determine which voxels intersect. As demonstrated before, it is impractical to check every pair of voxels between the two objects. To solve this problem, a hierarchy of bounding spheres was combined with a spacial-partitioning algorithm called an octree. An *octree* recursively partitions a space into octants, creating a tree structure which has eight children per node[15].

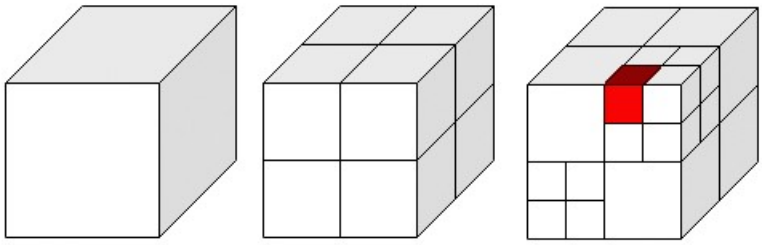

Figure 6: Recursive Subdivision Using an Octree
Source: http://procworld.blogspot.com/2010/11/space-warps-and-octrees.html

One of the advantages of an octree is that it allows for very fast rejection of collisions i.e. it can quickly determine if two objects do not intersect. This is because, if two objects are not even in the same octant, they cannot possibly intersect. If they are in the same octant, we recursively subdivide the octant until either both objects are in different octants or we have subdivided down to the level of one voxel i.e. the smallest unit. Thus, to determine if objects intersect, we can use an algorithm such as this:

```
start at the root node of the octree


while the unit of subdivision is greater than one voxel:
```

```
      find which octants obj1 and obj2 occupy
      if they occupy the same octant:
            subdivide the octant into eight smaller octants
      else:
            return "no intersection"

return "intersection"
```

This allows for a O(log n) intersection checking algorithm[15]. However, there are several subtleties. First, objects can be rotated. To be efficient, the octree would need to be axis-aligned. This means that every time the object is rotated, the octree has to be rebuilt. Second, we need to determine every pair of voxels that intersect between the objects. To solve this problem, I created a custom hybrid recursive algorithm. Around each node of the octree, a bounding sphere is placed. The algorithm then proceeds as follows, assuming the intersection is between the terrain and a cutting tool:

```
let node_terrain be the current node in the terrain (starting at the root node)
let node_cut be the current node in the cutting tool (starting at the root node)

let intersect_list be the list of all intersections of voxels between the terrain
      and the cutting tool.

def find_intersections:
      if node_terrain and node_cut have bounding spheres that intersect:
            if node_terrain has a subdivision level down to 1 voxel:
                  if node_cut has a subdivision level down to 1 voxel:
                        add node_terrain and node_cut to intersect_list
                  else:
                        for each child node CN in node_cut:
                              find_intersections between node_terrain and CN
            else:
                  for each child node CN in node_terrain:
                        find_intersections between CN and node_cut
      else:
            return
```

The idea behind this algorithm is as follows. If the bounding spheres of the current nodes intersect, subdivide the terrain until it gets down to the scale of one voxel. If this still intersects the cutter, subdivide the cutter until it reaches the scale of one voxel. If this happens and they still intersect, add the current terrain node and current cutter node to the output list. This finds every pair of intersecting nodes without having to start at the top of both trees for each intersection pair.

Deleting Voxels

Surprisingly, deleting a voxel turned out to be a fairly complex problem for a number of reasons. First, there is the issue of deleting the polygons that compose the voxel. When the polygon mesh was created, the data was uploaded to the graphics card via OpenGL[10]. Thus, this data no longer resides in main RAM. To edit it, it is necessary to ask OpenGL to map the vertex and color data into RAM, which can cause GPU pipeline stalls[10]. Unfortunately, this is only part of the problem: deleting a triangle from the VBO is equivalent to deleting an element from a very large array. As hundreds of triangle scattered throughout the buffer may need to be deleted, requiring deletion from an array for each, this method is infeasible.

To overcome this, a modification to the graphics pipeline was made to allow for an alpha blending component. Invariably, there must be some communication between the CPU and GPU. Using this method however, we only have to transfer twelve bytes of data per triangle that needs to be updated. *Alpha blending* allows for polygons to be partially transparent by setting an opacity factor[16]. This, however, did not work as expected when the polygon was made entirely transparent. As it turns out, alpha blending is sensitive to the order in which the polygons are drawn, even if one of them is entirely transparent[16]. In the end, the only viable solution given the current implementation was to modify the fragment shader to discard pixels which are transparent. This has the unfortunate consequence of forcing the GPU to rasterize triangles even if their individual pixels are transparent. A better solution would be to create a geometry shader that can discard entire polygons based on their transparency, though this option was not implemented in this project.

The next problem to deleting voxels has to do with the voxels which are adjacent to the ones being deleted. As discussed previously, the face that two adjacent voxels share does not have any polygons

generated for it, as they would never be seen. However, when a voxel is deleted, it potentially exposes faces that were previously hidden; it is necessary to generate triangles for the newly exposed faces of the neighboring voxels. This creates a number of problems. First, it is necessary to keep track of which triangles belong to which voxels. When the triangles are generated from the voxel grid, all the triangles that were created are guaranteed to be consecutive in the VBO. Thus, the original representation maintained a range of triangles that belonged to each voxel. However, in creating these new triangles, they have to be appended to the end of the VBO; this means that all triangles that belong to a particular voxel are no longer consecutive. To solve this, the structure was modified to be a linked list of ranges for each voxel. Fortunately, this information is only accessed when deleting a voxel so the performance lost from using a linked list is minimal.

The second problem is that the VBO is statically allocated when the object is created[10]. This limitation is from the way OpenGL manages buffers and does not have a simple solution. However, one work-around is to create a buffer that is a few times bigger than what is needed for the object. While this wastes extra space, it does allow for the possibility of adding new triangles. Another potential solution is to split the object into multiple smaller voxel “chunks” and render them separately. When a chunk needs to be updated, it is small enough to simply recreate with the new triangles added. Unfortunately, VBO's are expensive to create and render; typically, it is recommended to minimize the number of VBO's that are created for efficiency[10]. Thus, in this project, the first method was chosen.

Destructible Terrain

Once collision detection and the ability to delete a voxel were in place, the next logical step was to create a system which allows for destructible terrain. Destructible terrain means that one voxel object, representing the terrain of the game engine, is able to have parts of it cut away or destroyed by having it be collided with a voxel “cutting tool”. This ability is extremely powerful, albeit difficult to implement efficiently. Many game engines today use what is know as an *elevation grid* to represent the terrain. At each point, an elevation grid stores the height of the terrain at that point. These values are then interpolated to create a smooth terrain. While quite simple, this system does not allow for the ability to create caves or other multi-leveled terrain, nor does it allow for destructible terrain because only a single Y value (elevation) is stored for each X and Z coordinate. Voxels, by their very nature,

allow for discrete units that can be removed and displayed individually. This makes them an ideal choice for destructible terrain.

In the engine, there are currently two objects which the user can interact with. The first is the terrain object, which is an object that is “destroyable” i.e. it can be cut. By default, this object is 64 x 64 x 64 voxels. The second object is the cutter, which can be moved around by the user and optionally destroy the terrain that it passes through. By default, it is 8 x 8 x 8 voxels. When the engine starts up, it will prompt the user for two formulas: one that describes the shape of the terrain, and the other that describes the shape of the cutting tool. The very powerful thing about voxels is that they can be completely arbitrarily shaped. Thus, it is possible to have a cutting tool that is e.g. a sphere, a cone, or any other shape that can fit into an 8 x 8 x 8 voxel grid.

The cutting tool's color can be changed as the engine is running by pressing the keys 1 – 5. These colors are currently hard-coded, but it would very simple to allow customizable colors. As discussed previously, deleting voxels potentially exposes new triangles; to make things look more interesting and visually distinguishable, these new triangles are created with the current color of the cutting tool. This allows for the possibility of creating colored tunnels.

Conclusion

Though this project is not yet a complete game engine, much was learned from the design and implementation of it; it has been learned that voxels are a very powerful technique for working with things such as destructible terrain. Overall, it is hoped that it will lay the foundation for future graduate work; in addition, it is hoped that the knowledge gained from this project and future work will allow for the development of a C++ voxel library. One of the hopes of any project is that other people will find it useful and applicable to the problems that they are trying to solve.

As for future work, one of the main visual features that the engine is currently missing is lighting. When the voxel objects are created, or when the cutting tool cuts away portions of the terrain, each vertex of each triangle is assigned random shades of a particular color. When rasterizing a triangle, the color of each pixel is calculated by interpolating between these colors. Thus, in the future, it is hoped to

add proper shading and lighting calculations to give a more realistic rendering.

Another current limitation of the engine is that the entire terrain has to be loaded into memory at the same time. This means that only terrains which can entirely fit into RAM can be displayed. As voxel objects have a storage of $O(n^3)$ where n is the width of the object, this is a severe limitation. One of the possible solutions for this problem is to use an octree to partition the terrain and a viewing frustum to quickly identify if a node is at least partially visible. The *viewing frustum* is a truncated pyramid composed of six planes (near plane, far plane, top, bottom, left, and right) that define the region of 3D space which is visible to the camera. Using a simple dot product can determine the signed distance from a sphere to a plane. If this is done once for every plane of the view frustum, it is possible to quickly determine whether the sphere is partially inside the frustum or completely outside of it. If we put a bounding sphere around each node, we can determine if that node needs to be loaded into RAM by checking if the bounding sphere is visible. If not, we can keep the node stored on secondary storage.

Additional Images

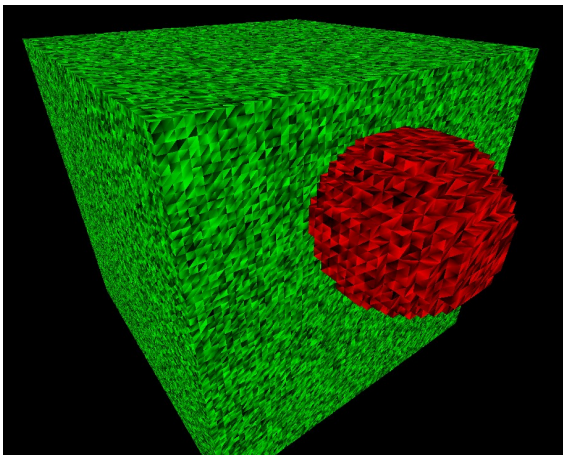

Figure 7: Cubular Terrain (green) and spherical cutting tool (red)

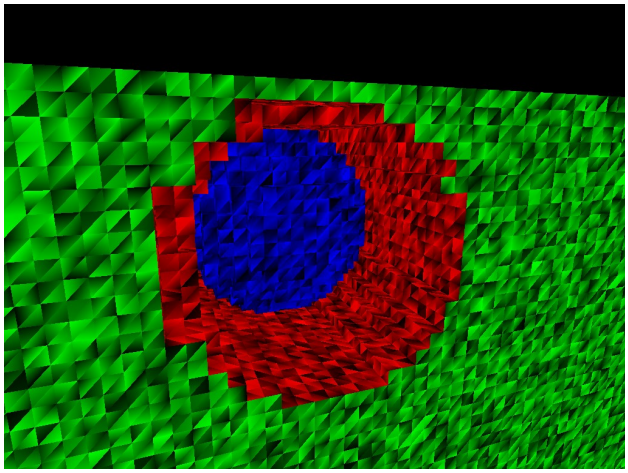

Figure 8: Cutting tool tunneling into the terrain and creating a red tunnel

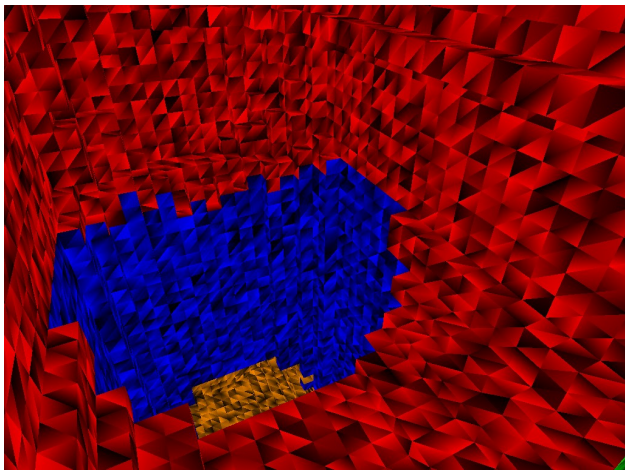

Figure 9: A multi-colored tunnel created by changing the color of the cutting tool

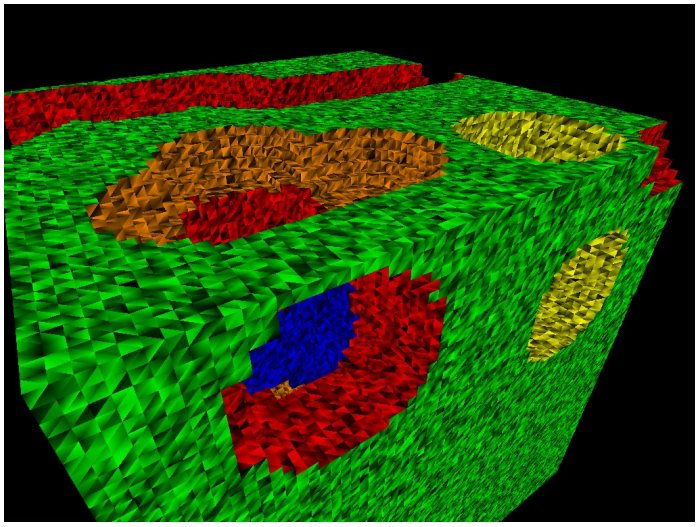

Figure 10: Multiple cuts made into the terrain

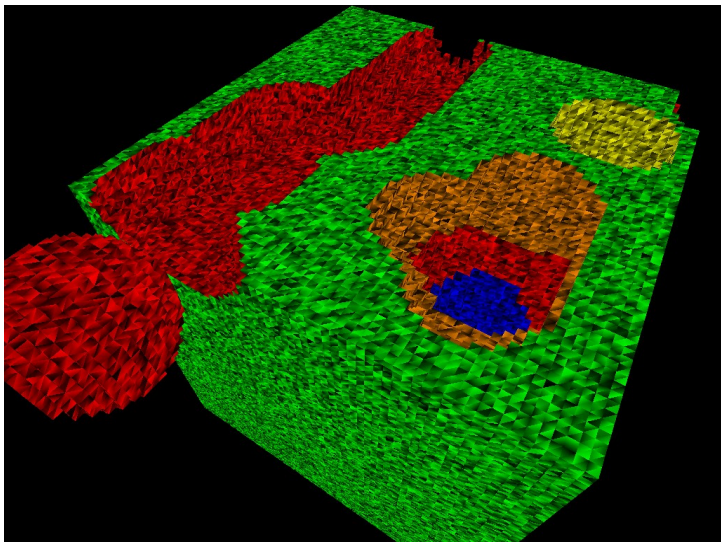

Figure 11: Figure 10 viewed from a different angle

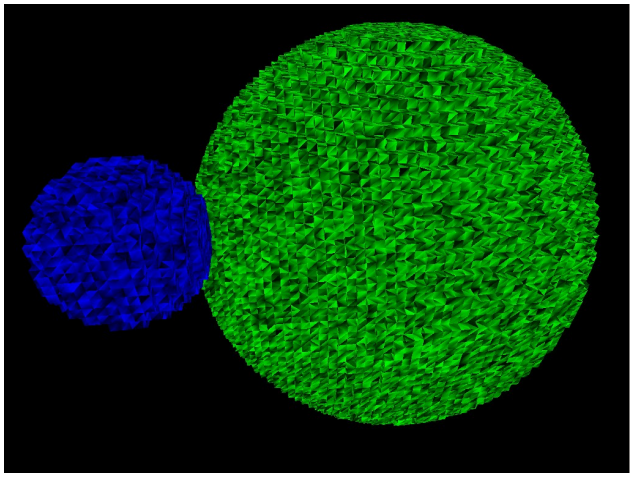

Figure 12: A spherical terrain and sperical cutting tool

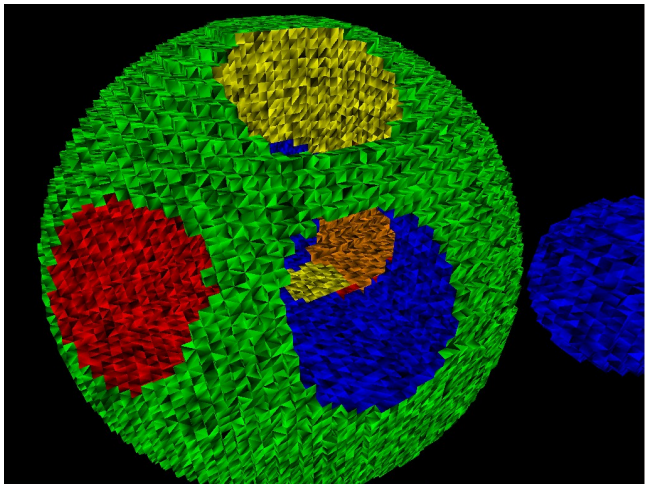

Figure 13: A spherical terrain with multi-colored cuts

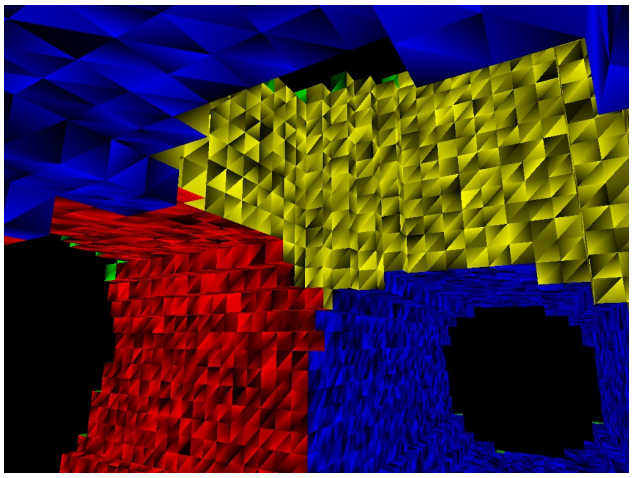

Figure 14: Figure 13 viewed from inside

References

[1]*Volume Graphics, IEEE Computer, Vol. 26, No. 7 July 1993*, Arie Kaufman et al.
http://labs.cs.sunysb.edu/labs/projects/volume/Papers/

[2]*Polygon Rasterization MIT Lecture Notes*, Andrew Vardy.
http://www.cs.mun.ca/av/old/teaching/cg/notes/raster_poly.pdf

[3]*Introduction to Design Patterns in C++ with Qt 2012,* Alan Ezust and Paul Ezust.
https://www.ics.com/designpatterns/book/vtable.html

[4]*OpenGL, Industry Standard for High Performance Graphics,* Khronos Group
https://www.opengl.org/

[5]*CMake,* Kitware
https://cmake.org/

[6]*Motorola 68k Programming Reference*, Motorola
http://www.nxp.com/files/archives/doc/ref_manual/M68000PRM.pdf

[7]*Two-Dimensional Arrays, Carnegie Mellon University*, Margaret Reid-Miller
http://www.cs.cmu.edu/~mrmiller/15-110/Handouts/arrays2D.pdf

[8]*Cache Memories, University of Berkeley, California*, Alan J. Smith
http://www.eecs.berkeley.edu/~knight/cs267/papers/cache_memories.pdf

[9]*Triangle Rasterization*, Brandon Llyoid
http://www.cs.unc.edu/~blloyd/comp770/Lecture08.pdf

[10]*Vertex Specification for OpenGL*, Kronos Group.
https://www.opengl.org/wiki/Vertex_Specification

[11]*Geometric Primitives*, Robert Sedgewick.
https://www.cs.princeton.edu/courses/archive/spr10/cos226/lectures/21-61GeometricPrimitives-2x2.pdf

[12]*Infix, Prefix, and Postfix,* Pete Jinks
http://www.cs.man.ac.uk/~pjj/cs212/fix.html

[13]*Collision Detection*, Brown Univeristy.
http://cs.brown.edu/courses/gs007/lect/sim/web/murat.html

[14]*Using Bounding Spheres*, Robert Dunlop.
https://www.mvps.org/directx/articles/using_bounding_spheres.htm

[15]*A Survey of Octree Volume Rendering Methods,* Aaron Knoll

http://www.cs.utah.edu/~knolla/octsurvey.pdf

[16]*Transparency, Translucency, and Blending,* Kronos Group

https://www.opengl.org/archives/resources/faq/technical/transparency.htm