5/1/2013

CubicEngine

By

Neel Rao - cs184-bo
Alex Wu - cs184-bx
Ashkon Soroudi - cs184-cb

Introduction

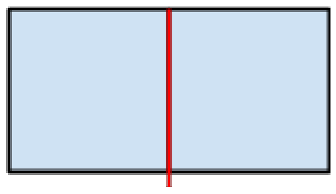
CubicEngine is a voxel-based terrain engine inspired by Minecraft. CubicEngine is capable of handling pseudo-infinite worlds with mountains, lakes, rivers, clouds, and trees. We also implemented rigid body physics with cubes and spheres. This allows the player to walk and jump around the world. The player can also throw spheres which bounce and interact with the terrain.

General Engine and Performance

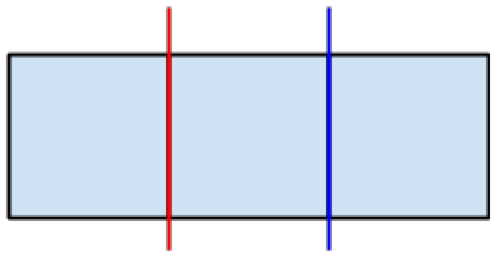
The hardest part for the general engine was rendering millions of cubes at a time with an acceptable framerate (>30fps). Our first approach, which was to simply draw every cube using glVertex calls only managed to draw in the order of 10^5 cubes with an acceptable rate. We needed one more magnitude of cubes to draw, so we added a lot more optimizations.

First, we used a 3D array of integers to represent the locations of the cube and the type (grass, water, air, cloud, wood, etc.). We then broke the world into chunks, which basically means a set of 3D arrays. We chose a chunk size of 16x16x128. The visible world is composed of NxN chunks, and the player is always in the center of this “chunk grid.” For example, if N is 3, then the player is always in chunk 1,1. When the player leaves this chunk to another chunk B, B is moved to the center and then all of the chunks are updated such that B is the center of the NxN grid.

Our first optimization was to not draw completely occluded blocks, meaning that in the array there is a block above, below, left, right, up, and down from the block. This is something that can be calculated very quickly. An generalization of this is to avoid drawing faces which are touching each other



For example, if the above two cubes are next to each other, we don't have to draw the face on the red line for both cubes. This is also a fast calculation. But, when we first implemented this, we found we were not getting huge FPS increases like we expected. The problem was with chunking.



Above, imagine that the blue line separates one chunk from another. In our original implementation, the faces on the blue line would be drawn because we optimized each chunk individually. To fix this, we built an abstraction over chunks which treated the world as one giant array, allowing each chunk to access blocks from neighboring chunks.

Our final optimization was to use openGL display lists. A display list is basically a pre-compiled openGL computation which give huge performance increases for static data. A display list takes time to generate, but once generated it can be re-used over and over again. We used display lists to cache the drawing calls for all of the cubes. We simply made each chunk a display list. This is why our chunk size is relatively small (16x16x128). Since, every time we leave the center chunk we generate chunks, we want smaller chunks so that the regeneration process (which includes compiling display lists), is relatively fast.

With all of these optimizations, we can achieve a resting framerate of around 30fps on a 2010 Macbook Pro with a 21x21 chunk array of 16x16x128 cubes, which totals to 14450688 cubes (14 million). 14 cubes aren't all drawn, since many faces are hidden by our optimizations.

Terrain Engine

Mountains, Hills, Valleys

Mountains, Hills and Valleys are all generated the same way via a modified version of perlin noise. What affects what will be generated is the height of the random numbers chosen and the frequency of interpolation. The way this is done and stored is with a height map. Each chunk has its own height map that it uses to generate its terrain. After the initial random values for the height map is generated, the rest of the values of the height map are filled in via cosine interpolation. These values are then used when setting the cubes so that we know what height we need to set them at.

These random numbers are generated with a pseudo-random number generator that takes a seed that the world seed concatenated with the hash of the chunkID. The chunkID is just the x and y position of the chunk . By doing this, we ensure that if we are given the same world seed, every chunk will generate the same height map every single time thereby creating the same world.

Clouds

The cloud algorithm was very simple. For each chunk, pick a random location at the highest height. Then, for N iterations, for each iteration move the pointer either in the x direction or z direction (y is height) with equal probability, and then render a cube block there. This is basically a random walk algorithm, and it produces interesting cloudy shapes. We chose N=50 for our project.

Trees

For each block on the top of a chunk, there is a random chance R that it will be a tree. If it is a tree, a random height is chosen, and wood blocks are placed until that height. At the top of the tree, a pre-set shape of leaves are drawn. Then, at random, a leaves are placed around the pre-set leave shape.

Water

Our initial water algorithm put random blocks of water in chunks, and then performed a ‘fill’ algorithm, sorth of like a bucket fill but with gravity. This had adverse effects, like turning entire mountains into water. The main problem was finding good places to start the water blocks.

Our final water algorithm was to turn any air block that was below X height into water. This filled our low valleys with water and gave a nice lake/river effect, so we chose this algorithm (it’s also fast).

<u>Caves</u>

We took out caves in our final implementation, but we still have code for it in Cave.java. The cave algorithm works using cellular automata rules. In 3D, there are 26 cubes around a given cube (around meaning that at least one vertex touches). The cellular automata rule was this.

For N iterations, do: Create a new copy of the world, with these rules. If in the original world, there are fewer than 13 cubes around a cube, then its copy will become an air block. If there are exactly 13 cubes around a cube, its copy takes the same type. If there are more than 13 cubes, then the copy becomes a solid block. Repeating this for a few iterations, like 3 or 4, starting from a random initial configuration of blocks, yields interesting cave structures. There are screenshots in the DVD of our experimentation with caves, but we left it out because we couldn't find a good way to mix the cave + mountains/hills/valleys because we can't dig or anything.

Pseudo-Infinite World Generation

We initially started testing terrain generation with one large chunk and a single height map. However, when we tried to figure out how to adapt this so that we can generate infinite terrain, it would not turn out right. Because of this, we went to our current system where we have a display array of size NxN that holds all of the chunks that we want to display. What chunks this array holds is dependent on the location of the player and the size of the array.

After we figured out this part, we needed to figure out how to get the terrain to generate infinitely and not have weird edge cases between chunks. The solution that we found was to interpolate between chunks and not at arbitrary step sizes. We are able to do this because each chunk has the ability to calculate the height map of any other chunk that is needs. This is because the seed used for a chunks height map is based off of its coordinates so all we need to do to find the seed of a chunk is to use the x and y coordinates of the chunk we want and figure out its chunkID. Once we have the chunkID, we can then use that to generate the height map that we need in order for us to correctly generate and set the blocks.

Method:

A		D		G	
B		E		H	
C		F		I	

The way this method works is based off the step size that we are given. The step size determines how many chunks we what to interpolate between. The “world” is then split off into “square grids” where the length of the sides of the grid is the number of chunks equal to the step size. So in the example above with a step size of 3, the “world” is broken up into grids of size 3(chunks) x 3(chunks).

Since every chunk in the grid ABDE has the ability to calculate what A, B, D, and E, what each chunk does is it figures out the corner positions for its own “grid”, calculates each height map for those chunks, and looks at the top left cube of that chunk. After doing this step, we now have the height of 4 cubes, one for each of the 4 corner chunks. Then, it interpolates between these 4 blocks in order to find the correct height of all of the in between blocks and then it sets the current chunk’s height map to the correct height, based off of the location of the chunk and the cube that it wants to set. When the interpolation happens, it only interpolates for the position that the chunk is. If we’re looking at the middle chunk between A and D, then it only interpolates for the values between .33 and .66. Also, because every single “grid” of chunks share edges, the edge values used for interpolation are the same. This way, all of the chunks will be seamlessly connected.

Player and Rigid Body Physics

We implemented basic player physics modelling the player as an aabb. We added gravity and a jump function. The player is only able to move or jump while he on the ground. To keep the player from falling into the ground, we added a ground state that would keep the player from moving in the y direction (falling through the ground) unless he jumps or moves off ground. Any collision in the x and z direction prevents the player from moving in that direction. We also added states for flying (press f key) and “no-clip” (ignore collision detection, press v key) for testing purposes.

We implemented basic rigid body physics using spheres and axis-aligned cubes. We used this Siggraph paper, http://www.pixar.com/companyinfo/research/pbm2001/pdf/notesg.pdf, as a guide to model the state of a rigidbody, as well as calculate collision response between rigid bodies. We used our own collision detection to detect collision for sphere-sphere collisions and sphere-cube collisions. At first, we had planned to implement cubes that could bounce in the world. We started by have a cube bounce off of a plane, and it rotated in a realistic way, although the end state (resting contact with the plane) had some problems. However, we found that non-axis-aligned cube-cube collision point detection was very difficult to implement, and we couldn't find any collision detection libraries in Java that would allow us to use their code with ours. Therefore, we decided to instead use spheres that would interact with the world made of cubes.

The player can spawn spheres with the r key that spawn in front of the player and move initially in the direction the player is facing. We used general rigid body physics, storing position, rotation, and momentum (linear and angular) as state variables. Force and torque are computed using collision response equations, and these are added to the momentum each step. This momentum is used to compute the velocities, which are then used to get the next state of the rigid body. In the case of resting contact with the top face of a cube (velocity relative to the collision normal is small), we add a force upward to counter gravity, and remove linear momentum in the y direction to avoid having the sphere sink into the ground. We also added an option for friction, where we just slow the balls speed if it's in resting contact with the ground.

Collision between spheres was fairly straightforward. There is a collision if the distance between the sphere's centers are less than their combined radii, and the point of collision is obtained by taking the normalized vector from one center to the other, multiplied by the radius. Sphere-cube collision were a bit trickier. First, we used a aabb around the sphere and did aabb-aabb detection in order to save time, since this is a very easy test. If this passed, find the closest point on aabb to the center of the sphere. This is done by clamping the distance from the center of the sphere to each side of cube's aabb. For example, if the sphere were directly above the cube, then only the y direction would be clamped down to the top surface of the box, and the resulting point would be the closest point. Then, test if the distance from this point to the center of the sphere is less than its radius. This will detect a collision, and the point will be the collision point. To get the normal of the collision, or which side of the cube the sphere collided with, we took the side with the largest clamped distance. For example, if the sphere were directly above the box, the largest clamped distance would be in the y direction downward, and thus it collides with the top of the box.

The last problem we had was with sphere rotations. We had very strange rotations that did not seem to make sense. However, we realized that this was due to innacurate time steps. According to the equations for collision response, the torque will be equal to some number multiplied by the cross product of r and n, where r is the vector from the center of the sphere to the collision point, and n is the normal of collision. However, for any sphere collision , these will always be parallel, and thus no torque will be added. Torque was still added because the collisions were not perfectly accurate with time steps. Realistically, rotations with spheres are largely caused by friction, which we did not model as a force. Therefore, we decided to instead use a different approach to model the sphere's rotation, by making it rotate in the direction of its movement. We took the axis formed by the cross product of velocity and the up vector, and used that as the axis of rotation, rotating an amount of degrees proportional to its velocity. This made the sphere's rotations look more realistic.