LECTURE 3
Announcements
Game/Engine Separation

• Minimally
  – src/engine and src/game, with appropriate classes in either, and no code in engine references code in game

• Better
  – src/engine/common, src/engine/voxel, etc…
README’s

• Global requirements specify to copy the rubric in AND leave notes about how you fulfilled each requirement
  – Don’t just give us a copy of the rubric

• Please remember hours taken on each project
  – Need these for future iterations of course
FPS

• No FPS requirement this week!
• Expect your game to run very slowly…
  — ~1-5 FPS
Minecraft

• Massively successful indie game (bought by Microsoft for $2B)
  – Adventure/exploration mode (survive in a world by collecting resources and fighting off monsters)
  – Creative mode (build to your heart’s content)
• Our version will support most, but not all, of their Alpha features
  – Up to you how much extra you add
Breakdown

• Week 1
  – World representation (voxel)
  – Terrain generation (common/game)

• Week 2
  – Rendering optimizations (common/voxel)
  – Collision detection (voxel)

• Week 3
  – Raycasting (voxel)
  – Infinite world (voxel)
  – Gameplay (game)

• Tons of room for creativity
  – Can spend hours on terrain generation alone (trees, mountains, caves, etc...)
Announcements

QUESTIONS?
LECTURE 3
The Voxel Engine
MOTIVATION

Voxel Engine
Spatial Organization

• Dividing your world into smaller groups
• Grouping entities into larger spaces
• Based upon some dimension
  – Could be abstract (think teams) – game specific
  – Usually physical – can do these in your engine
Why?

  – Collisions are $O(n^2)$ between all entities
    • $n(n-1)/2$ to be precise
  – Naively rendering all entities is slow
  – Even traversing everything to decide what to update/collide/render can be slow

• Environment representation
  – We don’t want a world made entirely of cylinders
  – Usually the environment has a very different representation than our entities
Simpler approaches

• Bounding box/volume
• Bounding hierarchy
Grids

• Commonly known as ‘Spatial Partitioning’
• Divide space into sections
• Uniform grids
Hierarchical Grids

• Grids within grids
Hierarchical Grids

• Commonly known as Spatial Partitioning

Quadtrees

Octrees
QUESTIONS?
What is a Voxel?

• Voxel = Volume + Pixel

• For our purposes, think of voxels as a 3D grid of cubes
Blocks

- Size 1x1x1 (a meter cubed)
- Efficient changeable terrain
- Position is implicit
  - Inferred from address in storage / position in array
  - One byte to specify type
  - Determine textures, transparency, passability, etc. from type
  - Maybe additional data? Up to you…
Chunks

- Store blocks in larger groups
  - 32x32x32
  - Should be able to change these dimensions with a single constant

- Compressed storage

- Uniform grid
  - Updates
  - Renders
  - (Eventually) collides
The Voxel Engine

• World structure (a set of chunks made of blocks)
• Rendering
  – Don’t draw hidden block faces!
    • Compute this at draw time?
    • Pre-compute this during chunk construction?
  – More optimizations next week
• Collision (week 2)
• Raycasting (week 3)
• Chunk Streaming (week 3)
The Voxel Engine

QUESTIONS?
LECTURE 3
Terrain Generation
(Common Engine?)
OVERVIEW

Procedural Generation (Common Engine?)
Procedural Generation

- Algorithmically generate your own maps
- Game side - experiment!
- Typically uses seeded random numbers
  - Calling `rand()` some number of times after being seeded by the same seed will always return the same sequence of numbers
  - The seed can be used to share or save the generated map
  - Used methodically to generate seemingly-hand designed content
- Different than randomly generated!
Constraint-based Generation

• Not just any random map will work
• Generated maps need to follow game-specific constraints
  – A dungeon crawler might require a path from entrance to exit
  – An RTS might require every area of the map accessible
  – Puzzles must be solvable
• Design your generation algorithm around your constraints
• Then consider soft constraints
  – What looks good, what’s fun, etc
Simple Generation Algorithms

- Perlin noise
- Spatial partitioning
- Exploring paths (random/drunken walk)
- Lots of resources online
  - Can you make your generation engine specific? (probably some of it)
Terrain Generation (Common Engine?)

VALUE NOISE
Perlin Noise

Named for its creator, this guy, Ken Perlin.

It’s a great way to make smooth, natural noise which can be used to create terrain, cloud patterns, wood grain, and more!

But you’ll probably use it for terrain...
We will be implementing a slightly more intuitive version called Value Noise.
What is noise?

- "Randomness"
- e.g. in one dimension our noise function will take an x coordinate (0-14) and produce a random value between 0 and 1
  - For terrain, think of this value as height
- By itself, our noise function is jagged and not useful
What is noise?

- In 3D, our noise function takes integer grid positions ((0, 0), (0, 1), (1, 0), etc.) and returns random height for each grid position.
- Noise function uses seeded randomness.
  - Pseudo-random values for different grid positions.
- Again, super jagged.
Making Terrain

- **Question:** How do we use noise functions to get something that looks like terrain
- **Wrong Answer 1:** Plug in each grid position to our noise function to get height for that position
  - As mentioned before, way too jagged
- **Wrong Answer 2:** Plug in every $n^{th}$ grid position to get height, and then use interpolated values for everything in between
  - We won’t get a lot of terrain detail, too spread out
- **Right Answer:** Think about wave functions
Wave Functions

Amplitude

Wavelength

frequency = \frac{1}{\text{wavelength}}
Wave Functions
Wave Functions
Wave Functions
Wave Functions

\[ 2\sin(2\pi) \]
Wave Functions

- For $\sin(x)$ we can change the amplitude and frequency using:
  
  $\text{amplitude} \times \sin(\text{frequency} \times x)$
Adding Wave Functions
Adding Wave Functions
Adding Wave Functions

\[ \sin(x) + \frac{1}{2} \sin(2x) \]
\[ \frac{1}{4} \sin(4x) \]
Adding Wave Functions

\( \sin(x) \) + \( \frac{1}{2} \sin(2x) \)

\( \frac{1}{4} \sin(4x) \)

\( \sin(x) + \frac{1}{2} \sin(2x) + \frac{1}{4} \sin(4x) \)
### Adding Noise Functions

<table>
<thead>
<tr>
<th>Freq.</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amp.</td>
<td>1</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{4}$</td>
<td>$\frac{1}{8}$</td>
<td></td>
</tr>
</tbody>
</table>

- For our noise function we can change the amplitude and frequency using:
  
  $$\text{amplitude} \times \text{noise( frequency } \times x)$$

- A frequency of 1 means that there is a spike at every single tile, while a frequency of 0.1 means that there’s a spike once every 10 tiles
  
  - Frequency of 0.1 maps a $10 \times 10$ grid to a $1 \times 1$ grid, so that there is only one noise value for the whole grid

- Add up different amplitude and frequency noise functions to get something nice
Interpolated Noise

• Great, so we can add noise functions
• But our noise functions only output values for discrete grid positions!
  – Depending on frequency, every tile, every other tile, every nth tile
• Now we want a continuous noise function that will always return the same value given the same seed and same (x,y) pair
• Let’s focus on the continuous part first, and the randomness given a seed/(x,y) part second
Interpolated Noise

• Most interpolation functions take three arguments
• $a$ and $b$, the value to interpolate between
• $x$, a value between 0 and 1
  – When $x$ is 0, function returns $a$
  – When $x$ is 1, function returns $b$
Interpolated Noise

• Option 1: linearly interpolate between points
• For any $a$, $b$, and $x$ on this graph:
  \[ v = a \ast (1 - x) + (b \ast x) \]
• This doesn’t look so great
Interpolated Noise

- Better option: cosine interpolation
  \[ f = (1 - \cos(x \times \pi)) \times 0.5 \]
  \[ v = a \times (1 - f) + (b \times f) \]
- Looks much better
- Slightly slower, but worth it for the results
Value Noise

- So what do we know?
- We know how to create a continuous noise function from a discrete one
- We know how to add big and small noise functions together to get more interesting noise
- So now we just need a noise function
A Good Noise Function

• What does our noise function need?
  – Given an (x,y) pair and a seed, returns the same value between -1 and 1 every time

• srand() only takes a single seed as an argument
A Good Noise Function

• No problem, I’ll just define some function that takes in x, y, and my seed and use that to seed frand()
• Good try, but then every point along that function that produces the same value will output the same result, and it will be obvious when looking down on your map
• Try experimenting with different noise functions!
Value Noise

class NoiseGenerator {
public:
    float valueNoise(Vector2 vec, float freq, float persistence, int num_octaves);
private:
    int _baseSeed;
    int _currentSeed;

    // feel free to make your own noise function
    float noise(Vector2 vec);
    float smoothNoise(Vector2 vec);
    float interpolatedNoise(Vector2 vec);
}

Value Noise

// returns a weighted average of the 9 points around (x,y)
// makes noise a bit smoother
float smoothNoise(int x, int y) {
    // four corners, each multiplied by 1/16
    corners = (noise(x-1, y-1) + noise(x+1, y-1) +
               noise(x-1, y+1) + noise(x+1, y+1)) / 16;
    // four sides, each multiplied by 1/8
    sides = (noise(x-1, y) + noise(x+1, y) +
             noise(x, y-1) + noise(x, y+1)) / 8;
    // center, multiplied by 1/4
    center = noise(x, y) / 4;
    return center + sides + corners;
}
Value Noise

// returns an value interpolated between the four corners surrounding the vector
float interpolatedNoise(Vector2 vec){
    integer_x = floor(vec.x);
    fractional_x = vec.x - integer_x;
    integer_y = floor(vec.y);
    fractional_y = vec.y - integer_y;

    // the four integer corners surrounding the float (x,y) pair
    v1 = smoothedNoise(integer_x, integer_y);
    v2 = smoothedNoise(integer_x + 1, integer_y);
    v3 = smoothedNoise(integer_x, integer_y + 1);
    v4 = smoothedNoise(integer_x + 1, integer_y + 1);

    i1 = interpolate(v1, v2, fractional_x);
    i2 = interpolate(v3, v4, fractional_x);

    return interpolate(i1, i2, fractional_y);
}
// returns a value between 0 and 1
// freq is the initial frequency of the largest “hill”
// persistence is between 0 and 1, determining how large each amplitude will be in relation to
// the previous one
float valueNoise(Vector2 vec, float freq, float persistence, int num_octaves) {
    total = 0;
    amp = .5;

    for(int i = 0; i < num_octaves; i++) {
        _currentSeed = _baseSeed + i;
        // so we use a modified seed for each octave
        total = total + interpolatedNoise(vec.x * freq, vec.y * freq) * amp;
        amp = amp * persistence;
        freq = freq * 2;
    }
    // adding waves can go outside of the -1 to 1 amplitude
    total = fmax(-1, fmin(total, 1));

    return total;
}
So what now?

- Once we’ve implemented the “noise” function in the above interface...
- Use valueNoise function to generate a 2D array of cells that each have a “height” which is between -1 and 1
  - We should normalize this to be between 0 and 1
- Now we can say what a particular floating point value means in terms of the y height of an (x,z) coordinate
More information

- Easily extendable into 3 dimensions
  - If passing in a 2D vector yields a 3D height map, then passing in a 3D vector yields a density map

- [https://web.archive.org/web/20160310084426/http://freespace.virgin.net/hugo.elias/models/m_perlin.htm](https://web.archive.org/web/20160310084426/http://freespace.virgin.net/hugo.elias/models/m_perlin.htm)
  - Incorrectly labelled as Perlin Noise (it’s actually value noise)
  - Plenty of good pseudocode to implement
  - Even has a 2D noise function! You might want to use it.
Procedural Generation (Common Engine?) – Value Noise

QUESTIONS?
LECTURE 3
Case Study – Minecraft 1
WORLD REPRESENTATION
World Representation

- World is capped at 256 blocks tall
- At the bottom (0), unbreakable “bedrock”
  - Generated to avoid a flat plane of bedrock
  - Sometimes resulted in “holes” allowing player to fall through the world
- At the top (255), an invisible barrier
- Other interesting heights
  - 62 = “sea level”
  - 127 = “cloud layer”
  - Various layers for different ore, such as coal, iron, gold, and diamond, as well as lava
Chunk Representation

- 32x32 wide in Alpha
- Changed to 16x16 wide in v1.2 for memory and efficiency reasons
- 256 blocks high
  - Covers the entire world
- Errors can result in erroneous chunks spawning
Block Representation

- **Block is 1 cubic meter**
  - Textures are originally 16x16, but can be modded for higher resolution

- **Block type is a single byte**
  - Allows for 256 block types (currently 175)
  - Recently changed to 12 bits w/ 4 bit metadata, allowing 4096 block types

- **Chunk stores lighting values per block at 4 bits per block**
Fancy Blocks - Gravity

• A select few, such as sand and gravel, are affected by gravity
• Limit # that move per update for speed
• Never cross chunk boundaries
  – Ours would have to…possible solutions?
Fancy Blocks - Liquids

• Water and lava
  – Shimmering textures
  – Impede/damage player on contact
• Flow using two types of blocks:
  – “Source” blocks are place-able
  – “Flow” blocks flow out of source blocks
• Flow 8 blocks in a horizontal plane
  – Reset if it travels downwards
• Tons of fun exploitables
  – Infinite water supply
  – Elevators
  – Perpetual motion machines
  – So much more (see Youtube)
Chunk Storage

• Players want to be able to store their worlds
  – Structures, items, explored areas, etc…

• But a fully-explored Minecraft world is ~140 quadrillion blocks
  – Impossible to explore them all, but you get the idea

• How did they compress that to a reasonable size?
  – Could spend an entire semester trying to answer this question
  – See http://minecraft.gamepedia.com if you’re interested – tons on chunk format, save format, etc…
Case Study – Minecraft 1

WORLD GENERATION
Terrain Generation
Terrain Generation
Terrain Generation
Biomes

• Different environmental areas in the world
• Blend smoothly using “rainfall” and “temperature” as the determinant characteristics
  – As opposed to “desert” chunks next to “swamp” chunks
  – Avoids sharp changes in biome
• Defined by block types, special areas/entities, and general structure
So how do they do it?

• We have no idea
  – It’s part of their secret to success, so online resources are slim

• What to start with:
  – 2D noise height-map (discussed previously)
  – Trees (seeded random numbers)
  – Drunken walk caves (like CS33, but 3D)
  – Snow-capped mountains (height check)

• If you want more:
  – Subtractive 3D noise terrain (very slow, difficult to tweak properly)
  – Multiplicative noise for your heightmap (good for mountain ranges)
  – Perlin “worms” (apparently how real MC caves are generated)

• REMEMBER — a proper implementation will be deterministic
  – Same seed, same exact block-for-block map
  – Easier said than done
LECTURE 3
Tips for Minecraft 1
Representing Chunks

- You’re going to have a lot of blocks that you need to keep track of
  - Make sure to dynamically allocate them!
    - Yes, this means use pointers. Deal with it
- Storing in a 1D array of size width*height*depth is faster than a 3D array of dimensions width, height, and depth
- So to get voxel at index \((x,y,z)\) you do:
  - \(\text{voxels}[x*\text{height}*\text{depth} + y*\text{depth} + z]\)
  - If you write a function to do this math for you, make sure you declare inline
- Consider what order \((x, y, \text{or } z \text{ first})\) is best
  - Depends on your generation
Managers

- In your common engine, your World keeps track of some Managers
- Your game specific code initializes a game-specific System, and registers it with your Managers
- Your world simply ticks and draws all its systems

```cpp
class Manager {
    public:
        virtual void onTick(float secs) = 0;
        virtual void onDraw(Graphics *g) = 0;
    private:
        List<Entity *> *entities;
};

class VoxelManager : public Manager {
    public:
        VoxelManager(List<Entity *> *entities, ChunkGenerator *generator);
        virtual void onTick(float secs);
        virtual void onDraw(Graphics g);
    private:
        ChunkGenerator *generator;
};
```
Managers

• How to use managers
  – Your VoxelManager will keep track of chunks and entities, and generate chunks
    • Later it will deal with collisions with entities and raycasting environment
  – Another possible manager, CylinderManager keeps track of entities, detects collisions between entities, and dispatches callbacks
  – Later on, GeometricManager will keep track of triangles in environment and entities, and deal with collisions
Leaky Textures?

• GL texture coordinates are between 0-1
• Neighboring textures can “leak” into each other from floating point imprecision
• Use filter GL_NEAREST when generating texture for both min and mag filter to prevent this
  – e.g. `glTexParameteri(GL_TEXTURE_2D, GL_MAG_FILTER, GL_NEAREST)`
LECTURE 3
C++ Tip of the Week
Representing blocks

• A block is (minimally) just a byte for type
  – Suggestion: typedef char as “blk” or something similar

• But what if we want more information?
  – Passability? 1 bool
  – Transparency? 1 bool
  – Faces to render? 6 bools

• So your Block will probably be a struct…
• On a 32-bit machine, a Foo instance will occupy 12 bytes
  - A char is 1 byte
  - A short is 2 bytes
  - So that’s 4 bytes, what gives?
• A “word” is 4 bytes, and each field is assigned one word
• We need to reduce memory usage if each chunk is 33k blocks…
  - That’s ~396 kB per chunk

// Foo is how many bytes?
struct Foo {
  char m2; // 1 byte
  char m2; // 1 byte
  short m3; // 2 bytes
}

Structured!
Struct Alignment

- C++ compilers allow you to “pack” struct
  - Great for network buffers, large datasets
- Use this for your Minecraft blocks!
  - Saves a ton of space, which in turn increases speed due to fewer cache misses
- Remember, structs can act like classes too!
  - Constructors
  - Destructors
  - Even methods!

```cpp
// gcc
struct Block {
    // Your block values
} __attribute__((packed));

// msvc
#pragma pack(push, 1)
struct Block {
    // Your block values
};
#pragma pack(pop)
```
PLAYTESTING!
To the Sunlab!