LECTURE 3
Announcements
Warmup2 Feedback

• Great work! Warmup is done!
  – 1/4 of the way done with the course (kind of)
• You all managed to make some really cool games
Testing on Department Machines

• Please do this!
• We’ve graded several projects that don’t compile – probably because things happen slightly differently on your personal computer
• We won’t deduct points for Warmup1 and Warmup2 if we can fix these problems quickly
• We will deduct points for future projects
Late Handins

- Please email your grader when you hand in a retry
  - Helps us get to it faster / get you feedback faster!
  - Prevent the snowball!
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
  – Voxel game representation
  – Terrain generation
  – Rendering optimizations

• Week 2
  – More rendering optimizations
  – Collision detection

• Week 3
  – Raycasting
  – Infinite world
  – Gameplay

• Tons of room for creativity
  – Can spend hours on terrain generation alone (trees, mountains, caves, etc...)
QUESTIONS?
LECTURE 3

Voxel-Based Games
MOTIVATION

Voxel-Based Games
Organizing Objects

• What do we mean by organization?
  – Grouping objects by their position
    • Things that are close
  – Grouping by abstract meaning
    • Object X is the parent of object Y which is the parent of object Z

• Why is organization important?
  – Performance
  – Structure / Representation
Why organize?

• A couple of examples ...
• Collision detection
  – Collisions are $O(n^2)$ between all entities
  – If we "organize" or "group" objects, can speed up collision checks
    • Check if larger groups are colliding first, then individual ones
• Environment representation
  – Often makes sense to represent environment hierarchically
  – Easier to code / maintain
Simple approaches

• Bounding box/volume
• Bounding hierarchy
Grids

- Divide space into sections
  - Group objects based on section they are in
- Example: Uniform grids
Hierarchical Grids

- Grids within grids
  - Create larger groups, then smaller groups within those groups
Hierarchical Grids

• Commonly known as Spatial Partitioning

Quadtrees

Octrees
Voxel-Based Games

QUESTIONS?
Voxel-Based Games

VOXELS
Organization in Minecraft

• We will have hierarchical Grid based on "voxels"
What is a Voxel?

• Voxel = Volumetric Pixel

• In minecraft, voxels can be thought of as …
Blocks

- **Size 1x1x1** (a meter cubed)
- **Attributes**
  - Transparency, passability
    - Maybe additional data? Up to you.
  - Texture
  - Position
- **Will NOT be GameObjects**
  - Too many of them
  - Probably represented as a small class or struct in your engine
  - Managed by ...
Chunks

- **Chunks** — grouping of blocks
  - 32x32x32
  - Should be able to change these dimensions

- **Responsibilities**
  - Updates
  - Renders
  - (Eventually) collides

- **WILL be GameObjects**
  - Probably have a component (or group of components) in your engine that make them chunks
  - Managed by VoxelSystem (next week)
QUESTIONS?
LECTURE 3
Rendering Chunks
Drawing Things

• We have a ton of different blocks stored in our chunks, so how should we go about drawing them?

• Perhaps intuition says to draw them one at a time, so let’s try that!
Attempt #1

• We’ll loop through each block, set the material, and then draw each block individually.
• But each time we draw a block, we have to send information from the CPU over to the GPU.
Attempt #1

• If our chunk is a measly 32x32x32, we’ll be doing this process for 32,768 blocks (for 1 chunk!)

• This means that for each of these blocks, a draw call needs to travel to the GPU for that block to be rendered
  – And this would happen on every frame
  – Sending data to the GPU is a huge bottleneck for drawing
Motivation

• THIS TAKES LOTS OF TIME
Motivation

• How can we avoid doing this each frame?
Attempt #2: Storing Shapes

• Solution: Create and store shape for entire pieces of static geometry
  – In our case, we’ll use it for terrain, per chunk
• Pack everything you need into a single Shape and draw it once
• Only a single draw call per chunk
Why bother?

• While fairly time-consuming to set up, the speed increase is incredible
  – Only a single draw call
  – Don't have to change texture for each block (even if they should be colored differently)
    • More on this later
How to do it

• Create each chunk shape:
  1. Create a Shape (see helper classes)
  2. Generate the Shape based on the chunk’s blocks
  3. Store the Shape

• When drawing terrain, iterate over each chunk:
  1. Draw the Shape
Generating the Shape (Faces)

- Want to store faces for every block in the Shape
- For each face, need to specify:
  - Vertices of that face
  - Triangles of that face
Generating the Shape (Vertices)

• For each vertex, you need to specify:
  – Position (self-explanatory)
  – Normal (the perpendicular to the cube face)
  – Texture coordinates (more on this next)

• Store all vertices in a vector of floats
  – 8 floats total per vertex
    • 3 for position
    • 3 for normal
    • 2 for texture
  – 4 vertices total per face
    • It's a quad
Generating the Shape (Triangles)

- Store all triangles in a vector of ints
  - 3 ints per triangle
    - Specify vertices in counterclockwise order
    - Int corresponds to position of the vertex in the vertex array
  - Two triangles per face
    - It's a quad
Textures

- Quick recap …
- Textures are basically just images
- Can use “texture coordinates” to specify what part of an image to texture a triangle with
  - $(0.0, 0.0)$ corresponds to upper left of image
  - $(1.0, 1.0)$ corresponds to lower right of image
- We specify the “texture coordinates” for vertices of triangle
  - Texture coordinates for in between points interpolated between these
Texture Atlasing

• When rendering chunks, we bind a single image (the texture atlas) which is used to texture all of the terrain
• Can specify texture coordinates for each face individually
• The texture coordinates are defined such that they map to subimages of the atlas
• ~10 fps boost guaranteed! (compared to binding an unbinding individual images)
Texture Atlasing

• You need to know the dimensions of your texture atlas first
  – Maybe the size of the textures too (if they’re uniformly sized)
  – Ours is a 256x256 image of 16x16 textures

• Subimages will likely be specified in pixels

• So we need to convert pixel positions to OpenGL texture coordinates
Coordinate Conversion

- To convert pixel coordinates to OpenGL texture coordinates:
  - $(x, y) \rightarrow \left(\frac{x}{\text{size}}, \frac{y}{\text{size}}\right)$
- Assume the same origin for both coordinate systems
- Example: convert point at bottom-left of grass
  - Texture size is 400x400
  - Point is at $(100, 300)$
  - $\left(\frac{100}{400}, \frac{300}{400}\right) \rightarrow (0.25, 0.75)$
Pseudocode

For a single chunk:

• Initialize the following:
  – A vector of floats that could hold ALL of your vertices
  – A vector of ints that can hold all of your triangles
  – A counter to keep track of the number of vertices
  – A Shape to hold the chunk’s shape

• For each block, for each face
  – Is the face visible? If so, add all vertices and triangles to your array, increment counter
  – Otherwise, skip the face

• Create a shape using the vertices and triangles
  – `std::shared_ptr<Shape> shape = std::make_shared<Shape>(vertices, triangles);`

• Do this every time the chunk is updated!
  – For now, only on load, but this changes next week
Tips

• Remember to use counter-clockwise order for triangle vertices!

• Use hierarchically layered functions!
  – `genChunkData` calls `genBlockData` for each of its blocks, and `genBlockData` calls `genFaceData` for each of block's faces
  – Will make debugging much less painful
OVERVIEW

Procedural Generation
Procedural Generation

- Algorithmically generate your own maps
- This will be 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)
In Minecraft

• We will use noise to determine terrain height for each (x, z) position in our world
Procedural Generation

WHITE NOISE
What is noise?

- Randomness
- e.g. From 0 to 14 take a random number between 0 and 1
- By itself, it is jagged and not useful
White Noise (2D)

- For each integer \((x, y)\) pair, generate a random (or seemingly random) value
White Noise (2D)

// returns a pseudorandom noise value from -1 to 1
float noise(int x, int y)
{
    int n = x + (y * 1777);
    n = ((n<<13) ^ n);
    return (1.0f - ((n * (n * n * 15731 + 789221) + 1376312589) & 0x7fffffff) / 1073741824.0);
}
Procedural Generation

VALUE NOISE
Value Noise

- Smooth white noise by taking an average of neighbors
- Turns white noise into something useful
// returns a weighted average of the 9 points around the Vec2i v
float valueNoise(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;
}
Procedural Generation

INTERPOLATION
Interpolation

• We can generate noise for integer (x, y) pairs
• What about things in between?
• Use interpolation
Interpolation

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

• Option 1: linear interpolation
• For values $a$ and $b$ and interpolation parameter $t$:
  • $f = a \cdot (1 - t) + b \cdot t$
Interpolation

- Option 2: cosine interpolation
- \( t' = \frac{1 - \cos(t \times \pi)}{2} \)
- \( f = a \times (1 - t') + b \times t' \)
- Slower, but much smoother
Interpolation

- Option 3: cubic interpolation
  \[ t' = 3t^2 - 2t^3 \]
  \[ f = a * (1 - t') + b * t' \]
- Similar to cosine
Interpolation

- Option 4: Perlin interpolation
- $t' = 6t^5 - 15t^4 + 10t^3$
- $f = a \times (1 - t') + b \times t'$
- Slightly slower than cubic
- Super smooth
Fractional Coordinates

• If our $x$ and $y$ aren’t integers …
• Just find the noise values the vertices of the unit square and interpolate
Fractional Coordinates

// returns the noise interpolated from the four nearest vertices
float interpolatedNoise(float x, float y) {
    glm::vec2 topLeft = glm::vec2((int) x, (int) y);
    glm::vec2 topRight = glm::vec2((int) x + 1, (int) y);
    glm::vec2 botLeft = glm::vec2((int) x, (int) y + 1);
    glm::vec2 botRight = glm::vec2((int) x + 1, (int) y + 1);

    float dx = x - ((int) x);
    float dy = y - ((int) y);

    float topNoise = interpolate(valueNoise(topLeft.x, topLeft.y),
                                    valueNoise(topRight.x, topRight.y), dx);
    float botNoise = interpolate(valueNoise(botLeft.x, botLeft.y),
                                   valueNoise(botRight.x, botLeft.y), dx);

    return interpolate(topNoise, botNoise, dy);
}
Procedural Generation

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...
Recall: Value Noise

• Smooth white noise by taking an average of neighbors
• Turns white noise into something useful
Perlin Noise

• Assign each vertex a pseudorandom gradient

Vec2f gradient(Vec2i vec) {
    float theta = noise(vec) * 6.2832;
    return new Vec2f(cos(theta), sin(theta));
}
Perlin Noise

• The noise value of each vertex is the dot product of its gradient and the vertex to the target point.
Perlin Noise

- Interpolate between the noise values of the four vertices (just like for value noise)
Procedural Generation

PERLIN NOISE VS VALUE NOISE
Perlin Noise vs Value Noise

- Value noise is easier
- Perlin noise has fewer plateaus
Procedural Generation

ADDING NOISE
Adding Noise Functions

• Our terrain still isn't interesting
### 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>$1/2$</td>
<td>$1/4$</td>
<td>$1/8$</td>
<td>=</td>
</tr>
</tbody>
</table>

- Adding noise functions together?
- Successive noise functions called "octaves"
  - Each has different "frequency" (how fast they change) and "amplitude" (how tall they are)
float octaveNoise(float x, float y, float persistence, int numOctaves) {
    total = 0;
    frequency = 1;
    amplitude = 1;

    for(int i = 0; i < numOctaves; i++) {
        // Add octave of noise
        total = total + interpolatedNoise(x * frequency, y * frequency) * amplitude;

        // Update frequency and amplitude
        frequency *= 2;
        amplitude *= persistence;
    }

    return total;
}
References

- Use this website as a reference for value noise:
  - May make some of the concepts / code discussed more clear
  - Another (harder) version of noise can be used, called perlin noise, which makes plateaus a bit smoother, and is more flexible.
A Note about Randomness

• None of the methods we've given you so far are random
• If you implement noise based on what we've given you, all of you will have the same terrain
• Tip: incorporate seeded randomness somewhere in your noise algorithm
  — Up to you where
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
• 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 used to be a single byte
  – Allowed for 256 block types
• 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
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)
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 Blocks

• Should NOT store...
  – Position
    • Position is implicit, based on position in a chunk's block array
  – Texture
    • Texture atlas shared between all blocks

• SHOULD store
  – Transparency
  – Passability
  – Some way of indicating what texture coordinates to use
Representing Chunks

- You’re going to have a lot of blocks that you need to keep track of
  - Store an array of ints / chars, where each char corresponds to a specific block to save space
  - Specific blocks initialized elsewhere in your game
- Storing in a 1D array of size width*height*depth is faster than a 3D array of dimensions width, height, and depth
  - To get block at index (x,y,z) you do:
    - blocks[x*height*depth + y*depth + z]
Chunk Contract

• You'll have a Game Object for each chunk, with a few different components
  – ChunkComponent holding your array of blocks
  – (Possibly generic) DrawableComponent to draw your chunk
  – Generic TransformComponent for your chunk's position.

• Again, the position of your individual blocks will be determined implicitly from their location in array combined with the chunk's location

```cpp
class ChunkComponent : public Component {
private:
    const int width = 32;
    const int height = 32;
    const int depth = 32;

    // Your array of blocks
    std::vector<char> m_blocks;
    std::map<char, Block> m_blockMap;
};
```

```cpp
class ChunkRenderComponent : public DrawableComponent {
public:
    void onDraw(Graphics g) {} 
private:
    Shape m_shape;
};
```
Representing Chunks (continued)

- **Game side**
  - Determine how chars map to specific blocks
  - Determine how chunk’s generate their blocks from noise

- **Engine side**
  - Array of blocks
  - Drawing
  - Transform
Using Noise (one approach)

• For each (x, z) column of your chunk
  – Generate a noise value (using octave noise) for chunkPos.x + x, chunkPos.z + z
  – Multiply the noise value by the max height of your world
  – Fill column of blocks up to that height
FPS

• No FPS requirement this week!
• If you implement only primary requirements (no storing shapes for each chunk) expect your game to run very slowly...
  — ~1-5 FPS
LECTURE 3

C++ Tip of the Week
C++ Templates!

- C++ templates are an extremely powerful tool for generic programming.
- Allows you to reuse the same code without losing any type specificity.
- Can be tricky to figure out though—it’s okay if you get lost!
#include <iostream>
using namespace std;

int square (int x)
{
    return x * x;
};

float square (float x)
{
    return x * x;
};

double square (double x)
{
    return x * x;
};

main()
{
    int i, ii;
    float x, xx;
    double y, yy;
    i = 2;
    x = 2.2;
    y = 2.2;
    ii = square(i);
    xx = square(x);
    yy = square(y);
}
template <class T>
inline T square(T x)
{
    T result;
    result = x * x;
    return result;
};

main()
{
    int i, ii;
    float x, xx;
    double y, yy;
    
    i = 2;
    x = 2.2;
    y = 2.2;
    
    ii = square<int>(i);
    xx = square<float>(x);
    yy = square<double>(y);
}
Template Classes

• In addition to have a templated function, we can have a entire templated class

• This is how things like vectors, maps, and the like are implemented

```cpp
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
    public:
        mypair (T first, T second)
            {a=first; b=second;}
        T getmax ();
    };

template <class T>
T mypair<T>::getmax ()
{
    T retval;
    retval = a>b? a : b;
    return retval;
}
```
PLAYTESTING!
To the Sunlab!