LECTURE 4
Announcements
Compiling on the department

• Clean and rebuild before handing in!
Modifying the Default Shader

• You may have noticed that the default shader makes things look...boring
  – Fragment shader res/shaders/shader.frag

• Offending lines

  \[
  \text{vec4 vertexToLight} = \text{normalize}(\text{vec4}(1,1,1,0));
  \]

  • Can treat light as positional, by making \text{vertexToLight} equal to light position minus fragment position (\text{position\_worldSpace})

  \[
  \text{float specIntensity} = \text{pow}(\text{max}(0.0, \text{dot(eyeDirection, lightReflection)}), 150);
  \]

  • Wow that exponent is big
Modifying the Default Shader

– Other

fragColor += vec4(max(vec3(0), vec3(1,1,1) * base_color * diffuseIntensity),0.f);
  – Change color of light by modifying `vec3(1, 1, 1)` be something else.

• Change contribution of specular, diffuse, and ambient by adding multipliers before the lines that modify `fragColor`

  fragColor += vec4(base_color*.3,0.f);
  – Change the last line so that the alpha component is 1, if you want to do anything with transparency
  – Will also need to call glEnable(GL_BLEND) and glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
Modifying the Default Shader

- Skyboxes!
  
  - [https://learnopengl.com/#!Advanced-OpenGL/Cubemaps](https://learnopengl.com/#!Advanced-OpenGL/Cubemaps)
LECTURE 4
Frustum Culling
(Common Engine)
Frustum Culling (Common Engine)

THE VIEW FRUSTUM
What is it?

- The volume of world objects that can actually be seen by the camera
- Shaped like a pyramid, bounded by:
  - Far plane (the “base” of the frustum)
  - Near plane (the “cap” of the frustum)
  - Field of view/viewport size (determine the “walls” of the frustum)
- This is what gets “unhinged” as the last of our 4 transformation matrices (CS123)
What we’re doing now...

- During `onDraw()`:
  - OGL transforms the entire scene based on the camera (to unit space looking down the $-z$ axis)
  - OGL renders everything using a z-buffer to figure out which object is closest to the camera for a specific pixel
    - That pixel is assigned that object’s color
    - Results in “Z-fighting” if depth values are close
- Can we avoid telling OGL to draw some things?
What we should do...

- Instead of sending everything to OGL, why don’t we avoid sending that we know won’t be drawn?
- What doesn’t need to be drawn?
  - Anything not in the view frustum!
- Only good if we can do this faster than OGL’s transformations and z buffer depth test
Extracting the View Frustum

- Frustum is defined by 6 planes
- Planes can be derived directly from our camera matrices
- glm uses column-major order, so use the coordinates given here to access the indicated cells

Projection matrix • view matrix

<table>
<thead>
<tr>
<th></th>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
<td>3,0</td>
<td></td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
<td>3,1</td>
<td></td>
</tr>
<tr>
<td>0,2</td>
<td>1,2</td>
<td>2,2</td>
<td>3,2</td>
<td></td>
</tr>
<tr>
<td>0,3</td>
<td>1,3</td>
<td>2,3</td>
<td>3,3</td>
<td></td>
</tr>
</tbody>
</table>
Extracting the View Frustum

- Plane equation is given by a 4D vector \((a,b,c,d)\):
  - \(ax + by + cz + d = 0\)

- The 6 clip planes of the frustum are defined below!

<table>
<thead>
<tr>
<th>Plane equation</th>
<th>-x</th>
<th>-y</th>
<th>-z</th>
<th>+x</th>
<th>+y</th>
<th>+z</th>
</tr>
</thead>
<tbody>
<tr>
<td>r3 - r0</td>
<td>r3 - r1</td>
<td>r3 - r2</td>
<td>r3 + r0</td>
<td>r3 + r1</td>
<td>r3 + r2</td>
<td></td>
</tr>
</tbody>
</table>

Projection matrix • view matrix
Frustum Culling Test - General

- Compute 6 plane equations
  - Should be updated whenever the camera changes!
- For each piece of scene geometry:
  - Does the entire shape fall behind one of the planes?
    • Skip rendering, it can’t be seen!
  - Does some part of the shape fall in front of all of the planes?
    • It could be seen, render it!
Frustum Culling Test - AABB

• AABB (axis-aligned bounding box)
  – Faces parallel to xy, xz, and yz planes
  – Defined by a position and dimensions (convenient!)

• Corners define all permutations of the most extreme values

• Rejection test: are all 8 corners behind any one plane?
  – For point \((x, y, z)\), reject if \(ax + by + cz + d < 0\)
Frustum Culling Test - Sphere

- **Sphere**
  - Defined by a position and radius (which can just be one of your dimensions!)

- **Rejection test**: is the center \((x,y,z)\) at least \(r\) units behind any one plane?
  - If \(ax + by + cz + d < -r\)
  - Planes must be normalized
    - Divide \((a,b,c,d)\) by \(\sqrt{a^2 + b^2 + c^2}\)

- You do not have to implement this!
  - Minecraft is all about cubes!
  - May be helpful later – spheres require a lot of vertices!
Implementation Notes

• Storing the r vectors
  – In the camera?
    • Only re-compute when camera changes
    • Not necessarily accessible in onDraw()
  – In the graphics object?
    • How to know when to re-compute them?
    • Always available in onDraw()

• Design decisions – yay!
Implementation Notes

• What should we cull?
• Individual blocks?
  – Fine-grained
  – Faster to just let OGL do the depth test
• Whole chunks?
  – Coarse-grained
  – Far fewer culling tests
• Non-environment entities?
  – Depends on # vertices
  – If we have AABB’s for them, depends on how many
• Required: per-chunk culling
Frustum Culling (Common Engine)

QUESTIONS?
LECTURE 4

Case Study – Occlusion Culling
Occlusion Culling

• Frustum culling renders everything in the view frustum
  – Z-test still performed

• Occlusion culling renders only things that can be seen from camera, not just those inside the frustum (e.g. cuts out objects obscured by other objects)
  – Avoids OGL z-test for those vertices

• Unity Demo!
LECTURE 4
Dynamic VBOs (Voxel Engine)
Dynamic VBO’s (Voxel Engine)

VERTEX BUFFER OBJECTS
OpenGL Review: VBOs

- Vertex Buffer Objects
- Big arrays of rendering data
  - Position
  - Normals
  - Texture coordinates
- Want to send vertex info to the GPU once since main bottleneck is on CPU -> GPU communication
  - Binding shaders
  - Binding textures
  - Setting uniforms
  - Drawing objects
- Stored directly in the GPU – just tell the GPU when to render which array of data and how to interpret it
Motivation

• Currently, each block is being rendered separately

• This means for each block, a draw call needs to travel to the GPU for that block to be rendered
  – This would happen on every frame

• you’re also probably sending some uniforms each frame to draw a number of quads/cubes
Motivation

• THIS TAKES TIME
Motivation

• How can we avoid doing this each frame?
Dynamic VBO’s

• Solution: instead of manipulating a single model many times, dynamically create vertex data for entire pieces of static geometry
  – In our case, we’ll use it for terrain, per chunk
• Pack everything you need into a single VBO and send it to the GPU once
• Only a single draw call per onDraw
• Avoid the use of uniforms as much as possible
Why bother?

• While fairly time-consuming to set up, the speed increase is incredible

• Per-chunk VBO’s + per-chunk frustum culling = ~50+ FPS with 200 chunks in memory!
  – This is roughly what’s required this week

• Free texture atlasing without any shader modifications (more on this later)
How to do it

• Initialize each chunk:
  1. Create a VAO/VBO.
  2. Calculate the vertex data for the chunk and put it into an array. (This is the hardest step!)
  3. Send the data to the VBO.
  4. Store the chunk’s VAO/VBO.

• When drawing terrain, iterate over each chunk:
  1. Bind the chunk’s VAO.
  2. Use `glDrawArrays` to render the chunk.
  3. Unbind the VAO.
Generating vertex data

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

• Remember: create triangles using counterclockwise winding order

• Design tip: hierarchically layered functions
  – `genChunkData` calls `genCubeData` for each of its cubes, and `genCubeData` calls `genFaceData` for each of its faces
  – Will make debugging much less painful
Texture Atlasing

• Because we’re dynamically creating the vertex data, we can specify texture coordinates for each face individually
• When rendering chunks, we bind a single image (the texture atlas) which is used to texture all of the terrain
• The texture coordinates are defined such that they map to subimages of the atlas
• ~10 fps boost! (compared to binding and unbinding multiple 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

• OpenGL texture coordinates go from 0.0 – 1.0
  – Origin is the top left

• Subimages will likely be specified in pixels

• So we need to convert pixel positions to OpenGL texture coordinate positions
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)
  \[\left(\frac{100}{400}, \frac{300}{400}\right) \rightarrow (0.25, 0.75)\]
What will we use these for?

• Per chunk!
  – Static environment objects
  – They can’t change (until next week)
  – Tons of vertices!

• Your voxel engine will handle rendering chunks as VBO’s
  – game code shouldn’t even know this is happening!

• Per-chunk VBO’s + per-chunk frustum culling = ~50+ FPS with 200 chunks in memory!
  – This is roughly what’s required this week
Pseudocode

For a single chunk:

- **Initialize the following:**
  - An array of floats that could hold ALL of your vertex data (including normals and texture coordinates)
  - A counter to keep track of the number of vertices (you'll need this later for drawing)
  - A VAO and VBO to hold the vertex data on the GPU

- **For each block, for each face**
  - Is the face visible? If so, add all vertices to your array, increment counter
  - Otherwise, skip the face

- **Send all of your vertex data to the VBO using glBufferData**
  - Data sent = #vertices * sizeof(one vertex worth of data)
  - One vertex is 8 floats (3 for position, 3 for normal, 2 for texture coordinates)

- **Do this every time the chunk is updated!**
  - For now, only on load, but this changes next week
QUESTIONS?
LECTURE 4
Collisions II
(Voxel Engine)
Collisions II (Voxel Engine)

COLLISIONS II
Last time on 3D Game Engines...

• if two shapes are overlapping
• Make them not overlapping anymore
• Called “non-continuous” collision detection
  – Objects teleport small distances – there’s no “continuity” from start to end
The new model

• Shapes are moving towards each other
• Only move them as much as they can go before colliding
• Never allows shapes to be overlapping
• Called continuous collision detection
Out with the old?

• This model is better for interacting with terrain
  – Typically depends on one of the objects being static
    • Chunks in this case
  – Falling through environment is typically game-breaking

• Previous model is better for entity interactions
  – Doesn’t know/care if objects are moving
  – “Missing” one collision probably isn’t a huge deal

• Why not use the same for both (like in 2D)?
  – Model determines the types of shape collisions that work best
  – Continuous won’t work well for both, neither will non-continuous
Best of both worlds

• How can we mix and match?
  – You world keeps track of “collision managers” that it ticks (and maybe draws)
  – Each “manager” shares the list of entities with the world
    • Entities are capable of returning whatever shapes the manager wants, or at least a position and dimension to create one
  – World is agnostic collision implementation
    • Your MinecraftWorld might use both a VoxelManager and a CylinderManager
Voxel Collision Detection

• Check for collision in all of the grid cells that an entity passes through in the next tick
• Move along x, y, and z axes separately
• If player would collide with a block during the next tick, move it to the boundary of that block instead
  – Otherwise, move the full tick value
• Player will automatically slide along surfaces
Example: X-axis Collision Sweep

- Search from cells containing `entity.bottom` through `entity.top`
- Player is moving in increasing x, check cells in that order

Move player as far as it can go
Move player back by epsilon (0.00001)
// Sweep test in x direction
void VoxelManager::xSweep(float dt, Entity *ent) {
    // direction of x motion, +1 or -1
    int step;
    // initialize start x, y, z, positions (must take dimensions into account)
    int startX, startY, startZ;
    // initialize end x, y, z, positions (must take dimensions, velocity, and dt into account)
    int endX, endY, endZ;
    // Loop and check for collisions
    bool collided = false;

    for(x = startX; x != endX; x += step) {
        for(y = startY; y != endY; y++) {
            for(z = startZ; z != endZ; z++) {
                if(isOccupied(x, y, z)) {
                    translateToCollisionPoint(x, y, z, entity);
                    collided = true;
                    break;
                }
            }
        }
    }

    // Move entity using x component of velocity, as if no collision
    if(!collided)
        moveEntityNormallyAlongX(entity, dt);
}
Implementation Tips

• Update entity position in the sweep method
  – Pass timestep – dt – to sweep method
  – Move entity location of collision if there is a collision, otherwise use
    • entity.pos[i] += entity.vel[i] * dt
      – i is the dimension that you are sweeping along
    – Can still update velocity in entity class, but you should no longer update position (or it will be update twice)

• Need to sweep in X, Y, and Z, but implement the y-sweep first
  – Easier to debug horizontal movement if you can stand on a floor

• If a collision is found, move the player away from the point of collision
  – Along surface normal multiplied by small epsilon
Implementation Tips

• Be careful at chunk boundaries
  – Player may be colliding with blocks in a different chunk

• Use your 3rd person camera to debug
  – Visualize the player’s AABB to *see* collisions

• Don’t be afraid to restart
  – The actual code isn’t too long (can be as little as \(\sim 20\) lines for each axis’ sweep test)
  – Lots of places to be off by 1 or epsilon
QUESTIONS?
LECTURE 4
Tips for Minecraft 2
Dynamic VBOs

• When generating the vertex data, passing your counter by reference can be helpful
  – Passing by reference means that changes made to that argument will affect the variable that stores it
  – e.g. void genBlockData(int &counter) will also increment the counter for you

• Use glDisable(GL_CULL_FACE) to debug
  – Even if you’re winding order is wrong, chunk faces will be drawn

• Spend a good chunk of time on design!
Texture Atlasing

• Various options for specifying which subimage of an atlas to use
  – Row-column x-y, raw pixel coordinates, etc.

• We recommend adding room to your Block class so that it can specify which subimage to use
  – Bonus points: specify which subimage for which face

• Up to you to decide how to do this
LECTURE 4

C++ Tip of the Week
• Declares a constant

- `const int x = 5;`  // constant integer 5
- `int const x = 5;`  // same
- `int const *p = &x;`  // cannot change x (int value)
- `int * const p = &x;`  // cannot change p (pointer)
Const++

• Const member functions
  – void Entity::draw() const { … // can’t change Entity
  – Very, very good style

• Using const references in functions
  – void Entity::accelerate(const vec3 &acc) { …
  – Only const functions can be called on const references
  – Much cheaper than accelerate(vec3 acc)
LECTURE 4

C++ Anti-Tip of the Week
Let's say we have a class:

class Dog {
    public:
        Dog(int age) {
            m_age = age;
        }
        int m_age;
}

And a function that takes an instance of that class:

    // use const references because it’s good style
    void doggyBirthday(const Dog &d) {
        cout << "Dog just turned " << d.m_age << endl;
    }

This should be a compile error:

doggyBirthday(3);

...right?
Implicit Constructors

• If you pass a wrongly typed argument to a function, the C++ compiler will root through your garbage to see if it can somehow convert that argument into the correct type.
• In our case:
  – `doggyBirthday(3) → doggyBirthday(Dog(3))`
• Only works with constructors that take a single parameter.
• In our testing, it only worked when the function took a `const` reference.
• To avoid this happening by accident, label your single-parameter constructors with the keyword `explicit`.
  – Hooray for more boilerplate!
WARMUP2 PLAYTESTING!

Don’t forget to sign up for Minecraft2 design checks (Now Thursday, Friday, and Saturday)!