CSC 165 – Final Exam Study Guide

Texture Mapping: Textures use their own 2d coordinate system with (0, 0) being the lower left and (1, 1) being the upper right. The X axis is typically called S and Y, T. Each vertex is programmatically assigned a texture coordinate. The rasterizer interpolates these values to "fill in" the pixels.

Skybox Vs. Skydome: A skybox is created using a cube map. This has images (of equal size) for all six faces of the cube. This cube surrounds the camera and moves with it. In contrast a Skydome uses a single "fish-eye" texture that surrounds the player as the top half of a sphere. The former is more common.

- **Skybox issues:** Takes 6 textures and generally requires 6 texture units. Can be hard to build and can show artifacts and seams/distortions.
- **Skybox Advantage**: Less vertices, more commonly supported
- **Skydome issues:** Great number of vertices, image applied must be processed in a specific manner
- **Skydome Advantages:** Single texture and therefore occupies a single texture unit.

How a Skybox Works: A skybox is generally implemented using two tricks:

1. Move the Skybox with the camera (position) but do not rotate it with the camera. Translate the box to the camera's location each time before drawing.
2. Use HSR (Hidden Surface Removal) implemented via the Z-buffer algorithm. This algorithm works by clearing the depth buffer to max depth, disabling testing/updating, drawing the skybox first, then re-enabling depth testing. This results in the skybox having "max-depth" and all objects drawn after will appear closer.
    - Z-Buffer Algorithm

```
If (pixel.z < depthbuffer[x, y])
    colorBuffer[x, y] = pixel[r, g, b];
    depthBuffer[x, y] = pixel(z);
```

Render States: States assigned to an object tell the game engine how to render the object. There are multiple states supported. Texture state tells the renderer to apply a texture, Z-buffer state tells the renderer to draw it like a skybox and transparent state draws the object transparent.

Height Map: A grid of values that represents heights (higher values = higher, lower values = lower). Grouped into triangles for translation.

Image-Based Height Map: An image where the corresponding color of a pixel represents the height of the world terrain. Black = low height (0) and White = high height (1).

- Easy to create, translated into height maps.

Hill-Raising: Create terrain by choosing a random point and radius. The algorithm then raises a hill with the chosen radius at this point. This is repeated for arbitrary iterations.

- **Algorithm for a single hill at (x1, z1) with radius r:** $y(x2, z2) = r^2 - ((x2 - x1)^2 + (z2 - z1)^2)$

Normal Maps: Normal maps make terrain (generally image-based) respond to lighting. This is an image file that holds the offsets to normal for the given heightmap. The values are generally stored as offsets from vertical relative to the plane tangent to the surface with the Z (or B) coordinates set to 1.0.

Following Terrain Height: A camera or an avatar can follow the terrain by getting the world X, and Z coordinates and retrieving the appropriate height from the height map object. The Y translation of the object is then adjusted accordingly. In this manner you can follow the terrain. Optional, you could also adjust angle based on neighboring heights.

Building Islands: An "island" can be built into the terrain by forcing all map values within the same boundary to be equal, by using hill-raising with the center forced to lie within the lake boundary or by adding an intersecting ground plane. The last is the simplest.

OBJ File Structure: The lines labelled ***v*** contain single vertices. The lines labelled ***vt*** contain texture cords (in S and T) for the corresponding vertices. Lines that start with ***vn*** contain coordinates for the normal vector for that specific vertex. Lines that start with ***f*** contain 3 tuples with 3 values each. Each value in each tuple correspond to a vertex in the triangle. For example, ***2/1/1*** means use vertex 2, with texture cord 1, and normal 1 for this face. The set of 3 defines a triangle.

DCC Tool & Content Loader: A DCC (Digital Content Creation) tool allows you to make something that by hand would take days in minutes. A content loader takes a model defined in a file (such as an OBJ) and loads it as an entity in the Scenegraph.

Blender use in this Class: In this class we used Blender to create model meshes (OBJ) UV Unwrap so they could be texture, rig models, and then keyframe them to produce animations.

UV-Unwrapping: The process of UV-Unwrapping takes a model and unwraps it at the seams to display the model in 2D. This produces an "unwrapped" image file that you can export and texture. The model is then saved with the corresponding texture cords for each face defined in the texture.

Rigging: Rigging a model is the process of defining a bone/join structure and associating a model's vertices with joints.

Keyframing: The process of creating a series of poses that define an animation. The movement between these poses is interpolated. These keyframes are saved to the model.

Keyframe Interpolation: If keyframe interpolation is done in the DCC, all frames of the animation must be saved in the model's file. If the interpolation is done in the game engine, the model files will be smaller. However, this also makes the game engine more complex.

- Essentially find the missing keyframe: **frameTime = frameNumber / fps**
- Select the nearest keyframes based on the current frameTime and interpolate position and rotation based on those keyframes

Linear Vs Non-Linear Interpolation: Linear interpolation for skeletons generally does not look right. For example, a bouncing ball with a straight path would be wrong, or a rotating arm that gets shorter as it points down. As a result, we want non-linear interpolation for bones.

2-Phase Collision Detection: A collision detection system that uses 2 phases. A broad phase where you select a pair of objects that might have collided (avoiding items that cannot collide. This is done as quickly as possible). Then using a narrow phase, we you closely examine those objects selected in the broad phase to see if they collided.

SAP (Sweep and Prune): Used in the broad phase collision detection. It moves along the axis noting the boundaries of objects overlap. It then sorts these boundaries by position on the axis. Any borders that are within each other could be a collision. These are examined in the narrow phase.

Physics Engine Role: A physics engine integrates the often complex laws of physics into a game. A physics engine is essentially a collection of solvers that are separate of game play. In use, you model your world in the physics world and run a simulation to see what would happen according to the laws of physics.

Setting up a Physics World: A physics world is setup by defining bounding shapes for objects that you want to have physics. You then update those objects based on the position and rotation of the physics object.

Physics Constraints: A constraint that can be associated with a physics object that limits its movement/rotation in some manner. Examples include a ball and socket, hinges, and sliders.

NPC and NPC Control: NPC's are characters controlled by the computer. An NPC controller generally resides on the client or server, or even a separate server. An NPC controller incorporates some algorithm to move the NPC. Controlling an NPC sometimes takes up to 50% of the game update time.

Server Vs. Client NPC Control:

- Server must frequently send update to clients about the NPC, clients must send information back when they interact with the NPC. NPC is easily synced across all clients.
- Clients control the NPC, or a single client has master control of the NPC. This requires more manual synchronization between clients through the server.

Purpose of AI in NPC's: AI's in NPC's give them intelligence to react to player actions. With AI an NPC can be an ally, a bystander, or an enemy.

FSM: Finite State Machines are often used to control NPC's. To make one you must implement code that manages the state, and code that executes depending on the state the NPC is currently in and code that switches states when necessary. These solutions are generally very flexible but specific to the game in question.

Simple Path-Finding: Or called Crash and turn. Moves directly to the target. If an obstacle is hit move parallel to that object until it is not blocked. Not optimal but generally good enough and cheap to process.

BOIDS: A simulated path/flocking algorithm that can coordinate a large amount of NPC handled as a single entity. Each BOID has simple attributes (speed, heading, position) and run a simple program. Consisting of 3 parts….

- Separation: Do not collide with each other
- Cohesion: Stay close to each other
- Alignment: Move in the same general direction

These are computed for each BOID and added to the velocity vector which then in turn affects the BOIDS position.

Behavior Trees: A tree that has roots and leaves. Similar in some manners to a FSM but generally has a consistent modelling approach. It is a popular approach to AI in games.

- **Leaf Nodes:** Can either be condition nodes (returns true or false based on some check) or action nodes (typically change NPC or game state). These are game dependent.

- **Interior Nodes:** Can either be a sequence node (like AND) or a selector node (like OR). The root is considered a selector node.

Tick and Think: Goal, stagger the think frequency to improve performance. Tick phase is done frequently and is simple and follows predictable steps such as move the NPC. Think phase is less frequent and performs the full decision making (evaluate a behavior tree) often also includes a tick phase inside of it.

Sound Wave Sampling: Sample frequency is the number of samples taken over a period of time and bits per sample is the sample size. Increasing both values improve audio quality but also increase size.

3d Sound: The ability to place a sound in a position and orientation and play that sounds as if we are hearing it from a given position and orientation.

3d Sound API: Performs the necessary calculations for audio based on the sounds position and hearer's position. We use OpenAL in this class.

Distance Attenuation: Sounds are softer the further they are away.

Doppler Shift: A sound waveform sounds different depending on how the source is moving (away or toward the listener).

Audio Position: Sound should be varied in intensity in the hearer's ear based on the position and orientation of the audio source.

OpenAL breaks sounds into different components to improve flexibility (same sound in multiple objects). The ability to move a sound around and playing different sounds at the same listener (mix and match).

- **Source:** A source of sound generation in the world. Attributes includes position, orientation, and velocity.
- **Listeners:** An entity that hears sounds. Generally, one per context attached to a player. Has attributes for position and orientation.
- **Buffers:** Attached to sources and holds the audio data (sound files). They can be asked to play their audio from the source's perspective.

Sounds Effects Vs Background: Sounds effects exhibit the attributes of 3d audio while background music should always be consistent regardless of the hearer's position or orientation.

Motivations for Quaternions: More compact (stored as a 1x4 vector), scalar, not vulnerable to gimbal lock, and easier to interpolate for smooth rotations.

Euler Vs Quaternion Format: Euler angles are stored as homogenous 3x3 matrices. They are simple to understand and easy to combine. Quaternions are stored as follows: ***Quaternion = (w, x, y, z)*** where ***w*** represents the rotation angle and ***x, y, z*** represent the rotation axis.

Gimbal Lock: Gimbal lock is the loss of 1 degree of freedom in 3d space. Occurs when 2 of the 3 axes occupy the same plane. The simplest way to avoid is to use Quaternions to model 3d rotations as they are immune.

Quaternions for Interpolation: Useful for interpolation because they are scalar. There are two simple approaches. Interpolation is the process of finding quaternions along a path between two points

- **LERP:** Linear Interpolation
  - Follows a straight path between two points.

    - Does not produce uniform change
- **SLERP:** Spherical Linear Interpolation
    - Produces a spherical path between two points with uniform change.
    - Produces two paths (long and short path)
    - Works poorly for small angles (Use LERP for small angles)