CS50's
Introduction to Game Development

OpenCourseWare

Colton Ogden (https://www.linkedin.com/in/colton-ogden-0514029b/)
cogden@cs50.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

Lecture 9: Dreadhalls

Today's Topics

- Texturing
  - We'll talk about how to apply textures and materials to objects in our scene.
- Materials and Lighting
  - Unity supports different materials and lights, which we'll discuss today.
- 3D Maze Generation
  - We'll be using a very simple but effective algorithm to generate our 3D maze, which we'll represent using a 2D array.
- First-Person Controllers
  - For this game, we will be playing from a fist-person perspective.
- Multiple Scenes
  - Similarly to how we modeled different states in LÖVE2D using a StateMachine, we can do something very similar in Unity using SceneObjects.

- Fog
  - Fog and global lighting are what will allow us to create our desired atmostphere for our game.
- UI Components and Unity2D
  - We'll discuss how to create UI elements in the game, and how we can leverage Unity2D to create 2D interfaces that we can apply to our 3D game.

Downloading demo code

- github.com/games50/dreadhalls (https://github.com/games50/dreadhalls)

Texturing

- Last week, we only used one texture (the city background for our helicopter game).
- We can create textures and modify their properties within the Unity editor.
  - For example, if we're trying to render a small texture onto a large cube, we can choose to "stretch" it via the editor, which would produce a low-res look for the cube, or we could "tile" the texture until it covered the cube entirely, which would produce a more high-res look for the cube.
  - We can do this by setting the `Albedo` component of a GameObject to a particular material or texture
- For more complicated shapes, we can use what is known as "UV Mapping" as a strategy for texturing.
  - That is, we can map out the 3D body as a flat surface, and texture the 2D surface instead, with different regions of the 2D model corresponding to distinct parts of the 3D model.
  - The Unity editor has its own built-in mapping algorithm for texturing 3D shapes, so you don't have to worry about implementing the details yourself.
  - Do note, however, that the UV mapping is not automatically updated when you modify your 3D object (e.g., when you shrink or enlarge it), so you would have to remove and re-assign textures as you modify your 3D models in order to maintain their original resolution.

Materials

- catlikecoding.com/unity/tutorials/rendering/part-9 (http://catlikecoding.com/unity

/tutorials/rendering/part-9/)

- Check out the above link to learn more about using materials in Unity and what you can do with them!

Lighting

- catlikecoding.com/unity/tutorials/rendering/part-15 (http://catlikecoding.com/unity/tutorials/rendering/part-15/)
  - Check out the link above to learn more about lighting in Unity!
- docs.unity3d.com/Manual/Lighting.html (https://docs.unity3d.com/Manual/Lighting.html)
  - The above link specifically dives into the different types of lighting.
  - Highly recommended reading for learning more about what you can do with lighting in Unity.

Normal (Bump) Mapping

- Bump maps allow us to take a flat surface and simulate 3D contours ("bumps") on it without having to create the geometry to make that possible. In other words, they allow us to essentially "fake" the lighting of bumps/dents on flat surfaces.
  - en.wikipedia.org/wiki/Normal_mapping (https://en.wikipedia.org/wiki/Normal_mapping)
- This is highly applicable for making walls look more realistic in 3D games without needing to slow your game down with expensive computations.

3D Maze Generation

- If we open up our Dreadhalls game in the Unity editor, we can zoom out of our scene to see the entirety of the maze from a bird's eye view.
- However, this ends up not being too interesting, since we cover our maze with a roof and thus can't see inside of it.
- We can remedy this by tinkering with some of the settings. At the top of the editor, by going to Window->Lighting->Settings, we can edit our Scene.
- The first setting we'll tinker with is the Environment Lighting.
  - This is a type of lighting that is applied uniformly throughout the scene to change the ambiance.
- Currently, we have a murky green color selected for our Environment Lighting, but feel

free to change it and see the effects!

- Another important setting in our Dreadhalls game is Fog.
  - Rendering Fog is essentially as easy as checking a checkbox, but there are also additional specifications that can be modified, such as Fog color, density, and more.
- Going back to our original problem, we can modify the current settings so that our maze will be visible to us in our Scene.
- This is especially important when debugging. By changing the lighting and removing the roof (via a checkbox in the "Level Generator (Script)" section), we can easily examine our maze.
- As you can see, our maze is not terribly complex, but it gets the job done.
- We model it using a 2D array, with walls and spaces represented as boolean values (`true` and `false`).
- The top-most and bottom-most rows and columns are, of course, marked as `true` in order to represent the outer-most walls in the maze.
- The remaining interior will be what we can modify in order to generate our maze.
- To do so, we will choose a random element within the interior of the 2D array as a starting point, and then "carve" a path through the array via a series of "coin flips", randomly choosing whether to move up, down, left or right and by how many steps.
- This is by no means a "traditional" maze generator algorithm, but it is cheap and efficient for our purposes.
- To learn more about other maze generator algorithms, check out:
  - catlikecoding.com/unity/tutorials/maze (http://catlikecoding.com/unity/tutorials/maze/)
  - journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes (http://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/)

Character Controllers

- Unity has a built-in Characters package that can be imported from the Asset store in order to access pre-built character Prefabs, including an FPS Controller GameObject.
- Upon import, it is automatically placed in Standard Assets->Characters-> FirstPerson->Prefabs, and can then be dragged and dropped onto the scene (from within the editor) in order to assign it as your character's default camera.
- The FPS Controller itself is simply a kinematic capsule collider with a camera at the top, whose perspective can be controlled with the mousepad and whose position can be modified with the WASD keys.

    - If curious, the scripts for the FPS Controller are also included upon import. They are found in the Scripts folder within FirstPerson.
- As you can see, creating a first-person controller for your game in Unity is a process that takes about a minute or less.
- Additionally, there are many customizable features that can be applied to the FPS Controller (e.g., walk/run speed, step size, jump speed, footstep sound, mouse sensitivity, head bob, etc.)!

Multiple Scenes

- In order to load multiple scenes in our game, we take advantage of `UnityEngine`'s built in `SceneManagement` method.
- For example, in the `GrabPickups` script associated with our FPS Controller, we monitor for collisions. If we detect a collision with a GameObject having the `Pickup` tag, we play a sound and load a new scene using `SceneManager.LoadScene`.
- Similarly, in our Dreadhalls title screen, we have a script running (`LoadSceneOnInput`) to check for when the user presses "Submit" in order to start the game, which we load by again calling `SceneManager.LoadScene`.
- Of course, this presents a potential issue. When we reload our scene with `SceneManager.LoadScene`, we are destroying all existing GameObjects and re-generating new ones, including our audio sources.
- You can imagine that this might cause some strange behavior with our background music jumping around, which we don't want.
- Ideally, we'd prefer a seamless loop of background music, regardless of whether we're reloading the scene or not.
- Thankfully, Unity provides us with a `DontDestroyOnLoad` function to bypass this issue.
- However, we must take care not to re-instantiate a new audio source for the background music each time that we load the scene.
- To get around this new problem, we can instantiate our first background music audio source as a "Singleton" object, such that there can only ever be one.
- This behavior is implemented in the `DontDestroy` script, in which we essentially mark the first background music audio source as the main `instance` and subsequently destroy any additional background music audio sources that are instantiated upon loading a new scene.

Unity2D

- We've mostly been working with Unity's 3D features up until now, but Unity also provides functionality for 2D game development.
- This might be particularly useful in 3D games for creating title screens, for example.
- In our program, we use Unity2D to create our "DREAD50" title screen.
- This provides us with a very powerful engine for creating 2D scenes, with which we can interact very similarly to how we can interact with 3D scenes in the Unity editor.