Getting Started with Final Project: Github Repo & Starter Code

COS 426: Computer Graphics (Fall 2022)

Guðni Nathan Gunnarsson, Yuanqiao Lin, Yuting Yang

Agenda

- Final Project Logistics
- Set up Github Repository
- Starter Code

Final Project

- Proposals were yesterday Dec 8 - 3:00pm
- Presentations and Demos, Dec 14-15
- Written Report
  - Due Dec 16 (Dean's date)
  - Example outline available on project specs
- Check-in twice with your assigned TA
  - Sometime this week (or early next week!)
  - Before the presentation
- We recommend you start early!

Starting things - starter code

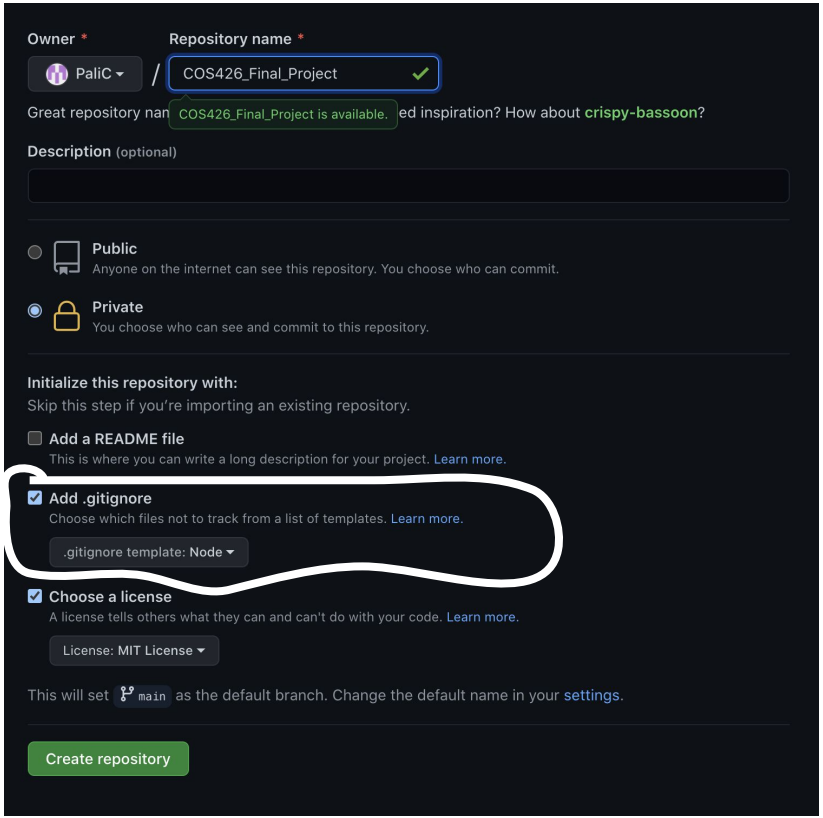
- Download the starter code
- Read the ReadMe and Try to Run It!
  - We will demo this
  - Mac User: "`Error: \`gyp\` failed with exit code: 1`"
    - try "`xcode-select --install`" to install xcode tools
    - or "`sudo rm -rf $(xcode-select -print-path); xcode-select --install`" to reinstall

Starting things - git

- Turn the seed into a repository on github or start your own one
  - Share it with your partner
  - If your new to git look at this cheat sheet
  - Add a .gitignore file
- Highly recommend using VSCode for git integration

Add a .gitignore file !!!!

- NPM installs packages locally for you.
- Make use of this and don’t push packages otherwise it’ll take forever to git push and pull



Very Basic Git Workflow

- Useful commands:
  - `git pull`
  - Make your changes to the code
  - `git add <modified file names>`
    - To add all changes you can just use '`git add .`'
  - `git commit -m '<commit message>'`
  - `git push`

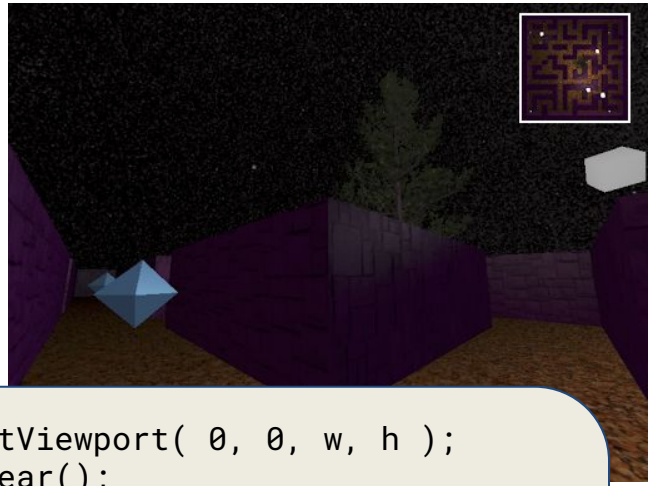
Final Project
- Optional Requirements overview -

Texture mapping (fairly simple)

- ThreeJS has built-in texture support.
- Make sure that your models have proper UVs
- You can also set up texture filtering.

Multiple views (intermediate)

- Several ways to achieve this. The canonical way is to render the scene twice per frame. To do this update the render loop.
- Swapping between views is simpler, so if you do that, try to use a novel viewpoint such as over the shoulder.



```
renderer.setViewport( 0, 0, w, h );
renderer.clear();

// full scene with perspective camera
renderer.render( scene, camera );

// minimap with orthogonal camera
renderer.setViewport( 0, h - mapHeight,
     mapWidth, mapHeight );
renderer.render( scene, mapCamera );
```

On-screen control panel (fairly simple)

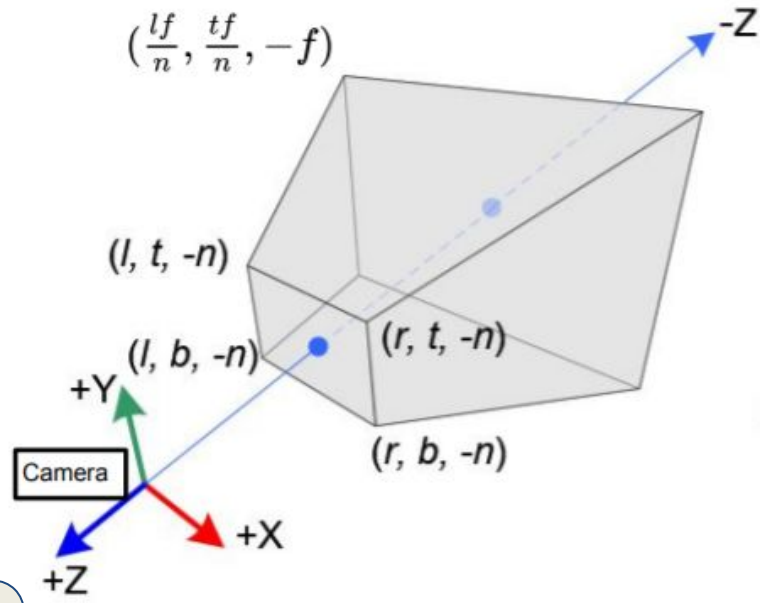
- The control panel is already present. See the gui documentation here. Set up the gui for your scene like so:

```
this.state = {
     gui: new Dat.GUI(), //Create GUI for scene
     rotationSpeed: 1,
     updateList: [],
};

//Add a control for rotation speed
this.state.gui.add(this.state, 'rotationSpeed', -5, 5);
```

View frustum culling (intermediate)

- Save on rendering time by never sending off-screen objects into the renderer.
- Iterate through all your game objects, and check if they are inside the frustum.
- Here is a helpful resource on view frustum culling.
- For full points, please refrain from using the THREE.Frustum type and have `frustumCulled` set to false for all objects.




```
//In the render loop, recursively check all elements
scene.traverse(function(element) {
     element.visible = frustum_check(element, camera)
});
```

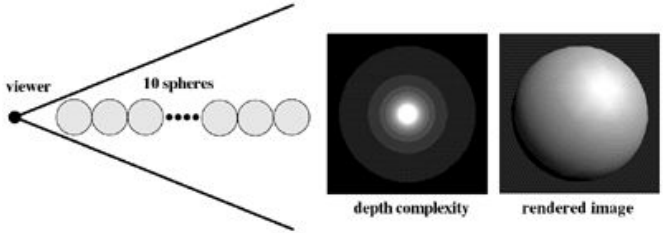
Level of Detail control (intermediate)

- Use lower detailed models when far away.
- You will need to take your models and make lower detailed versions with a program like blender or find models with built in LOD.
- ThreeJS already implements this feature, but for full points you will need to implement it yourselves.

```
//In the render loop, recursively check all elements
scene.traverse(function(element) {
     if (element.LOD !== undefined && element.visible)
     {
          const updatedLOD = getLOD(element, camera); //Make this function
          element.children[element.LOD].visible = false;
          element.children[updatedLOD].visible = true;
     }
});
```

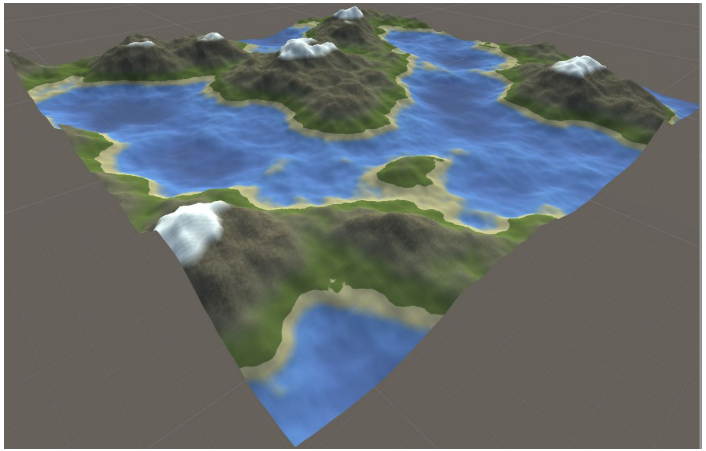
Occlusion Culling (advanced)

- Hide objects that are occluded by other objects.
- Different methods include:
  - Visibility Test
  - Hierarchical Visibility
  - Hierarchical Occlusion Map
- See the gamedeveloper website to get started.



Procedural and physically-based modeling (intermediate - advanced)

- Using controlled noise (think perlin noise, fBM) to generate terrain or particles.
- Classic use-case is generating a height-map. For full effect, use textures or water to communicate the heights.
- Here is a simple tutorial for heightmaps.
- Plenty of other possible uses.
- This works well when combined with custom GLSL shaders.



Collision detection - (advanced)

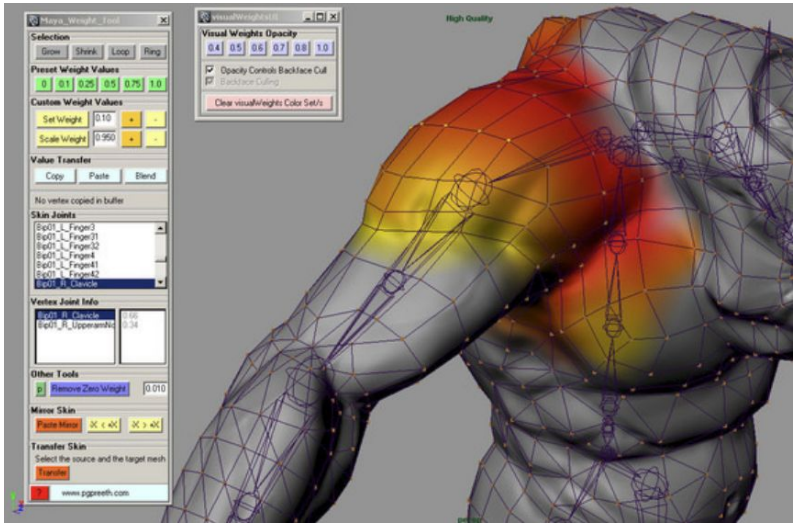
- Almost any proper game will need collisions.
- You can use a library, but for full points implement them yourselves.
- Your collisions do not need to be complicated. Simple box-box, sphere-sphere or box-sphere collisions may be sufficient for many games.

Simulated dynamics - (advanced)

- Modeling dynamic behaviors and physics for objects in the 3D world.
- Works well when paired with collision detection.
- For example, the wheels of a vehicle might react realistically as they move over rough terrain (e.g. spinning, springs)
- Or a door might swing open differently depending on the force exerted by the player. (e.g. shoot the door vs or tapping on it).
- This could be an extension of assignment 5.

Skinned Characters - (advanced)

- Given a moving character model, we want to deform the skin so that it aligns with the skeleton.
- You will need animated, rigged characters. (see for example mixamo.com or this 3js example)
- Although ThreeJS has already skinning, you will have to make your own implementation for full points here.



Vertex or fragment shaders (fairly simple - advanced)

- Add custom shaders to implement custom effects or offload processing to the GPU.
- Useful for other types of advanced functionality, such as:
  - Procedural generation,
  - Texture mapping,
  - Non-photorealistic Rendering,
  - Animation (especially massively parallelized),
  - Particle systems,
  - Mirrors,
  - Portals,
  - Non-euclidean geometry,
  - and even the kitchen sink.

Advanced image-based techniques (intermediate - advanced)

- Billboarding - Sprites that always face the camera
- Environment mapping - Quick and dirty reflections
- Once again many of these techniques are already implemented in ThreeJS. So for full points here you will need your own implementation.
- However, you are allowed to use the CubeCamera to generate cubemap textures.
- Pairs well with GLSL shaders

Environment mapping

Billboarding

Sound (fairly simple)

- ThreeJS has support for audio listeners and audio sources which may be used. You will have to build the infrastructure of playing the correct sound at the right time (e.g. with events).
- https://freesound.org/ is an excellent source for creative commons licensed audio - don’t forget attribution.

Networked multi-player capability (advanced)

- Multiplayer through a network.
- Can provide a lot of bang for your buck - it's always a good time to play with friends.
- Use the servicenet wifi network to be able to communicate.
- We recommend using websockets so the server can be programmed in any language.

Networked multi-player capability (advanced)

- Many possible architectures, but one is a simple broadcasting server.
  - Every frame (or tick) each player sends their data to the server. This would include player positions and the actions that they take. The server broadcasts to all the other players.
  - Each player reads the data from the server to update their game world.
  - One player is designated “the leader”. If there is a disagreement (e.g. who was killed first), then the leader is the tiebreaker.
  - Good for up to 4 local players, but scales poorly after that.

Game level editor (advanced)

- If your game features complicated levels, like a shooter or platformer, you might consider making a supplementary program to design and create levels.
- This application may be part of your game, a separate ThreeJS application, or any other graphical application.