PLUNGE - Project on Textures and Animations

Agam Kashyap
IMT2018004
International Institute of Information Technology, Bangalore
agam.kashyap@iiitb.ac.in

Saksham Agarwal
IMT2018065
International Institute of Information Technology, Bangalore
saksham.agarwal@iiitb.ac.in

Utkarsh Agarwal
IMT2018082
International Institute of Information Technology, Bangalore
utkarsh.agarwal@iiitb.ac.in

***Abstract*—This is a detailed report on our project PLUNGE, which utilises Textures, Animations and Lighting to generate a three dimensional scenery.**

***Index Terms*—Ambient, Directional, UV mapping, FSM, Animation, Camera, Skybox, Scene Graph**

I. Introduction

PLUNGE is our attempt of rendering a scene with player controls and interaction with other objects in the scene. We tried to create as realistic scenario as possible based on what we learned in our course- CS606. Utilising the ThreeJS library to provide us with enhanced functionality in WebGL, we introduced lighting, textures and animations. PLUNGE can be experienced at https://3dscene.netlify.app/

II. Method

A. Scene

Scene graph plays an important role in the complete scene, facilitating easier relative transformations for the objects. The scene graph for this project can be scene in Fig **??** Our scene consists of a 3D mesh consisting of buildings, creating an alley among them. In this alley, we have street lamps running along each side. Each of these lamps have a spotlight pointing in front of them. One of the towers in the scene has a Track-Light which tracks the movement of the player. We implemented this by utilising the `light.target` attribute and setting it equal to the player.

The scene itself is a skybox made by UV mapping of a set of images. This feature has enabled us to somewhat simulate the day and night scenario. So, to further enhance this, we changed the type of lighting present in the scenario. The Directional light present in the night scenario is a dark shade of purple(`0x210021`) while the light present in the day scenario is a very light yellowish color (`0xfdfbd3`) which is generally the colour of morning light. The street lights also cast less intense light in the morning adhering to the physics. We have an Ambient light of pure white color (`0xffffff`) and intensity of 0.1 present in the scene.

Problems: The issue came when I was trying to define the position of the spotlight in the scene. So the Mesh of lamp is actually a group composed of various children and grand children. In the beginning we made the Spotlight a child of one of the children of the street lamp's base object. But, we noticed that the spotlight was still positioned at the origin. After debugging, we realised that the creator of the mesh had manually defined the positions of each section of the lamp and the transformation matrices were all identity. Hence, we had to manually position the spotlight so that it comes in the correct position.

B. Cameras

We have three types of cameras - First Person Camera, Third Person Camera, World Camera. In the First person view, the camera is supposed to act like the eyes of the Player, with the same position as the head. In third person camera, the camera is positioned a little distance behind the player and instead of looking directly at the player, we make the camera look at a point few units ahead of the player. The World Camera has no association with the player. It is free to move and look anywhere in the scene.

Approach: To implement it, we started by building it for a cube based player first. The most challenging one was the first person camera, in that, it was difficult to come up with a proper formula for getting the camera to look at the right vector. To rotate the player - in initial case, cube - we were converting the mouse coordinates into spherical coordinates, which was then used to create a "LOOK DIRECTION VECTOR", which we feed to the player as `player.lookat(DirectionVec)`.

Now, in the initial stages we felt it would make our task easier if we make the cameras (First Person and Third Person) children of the player, since this would automatically make the camera move along with the object. But the issue it presented was in setting up the orientation for camera. Since we are already rotating the object using `player.lookAt()`, the camera must take into account that orientation and according to that align itself. This method would involve getting the current orientation of the object (can be done using `.quaternion`) and applying the transforms necessary for the camera. We wanted to make our project as realistic as possible, and so we decided to allow the camera to up and down as well. Since the movement of player is restricted along the longitude, we had to come up with an alternative approach.

So, we finally decided to not have cameras as the children of the Player. Instead, we compute the position the camera is supposed to be at with respect to the player, and also the point where it is supposed to look. We pass in these values to the camera, and hence prevent any hindrance like before. But this wasn't straightforward as well. We wanted the position of the

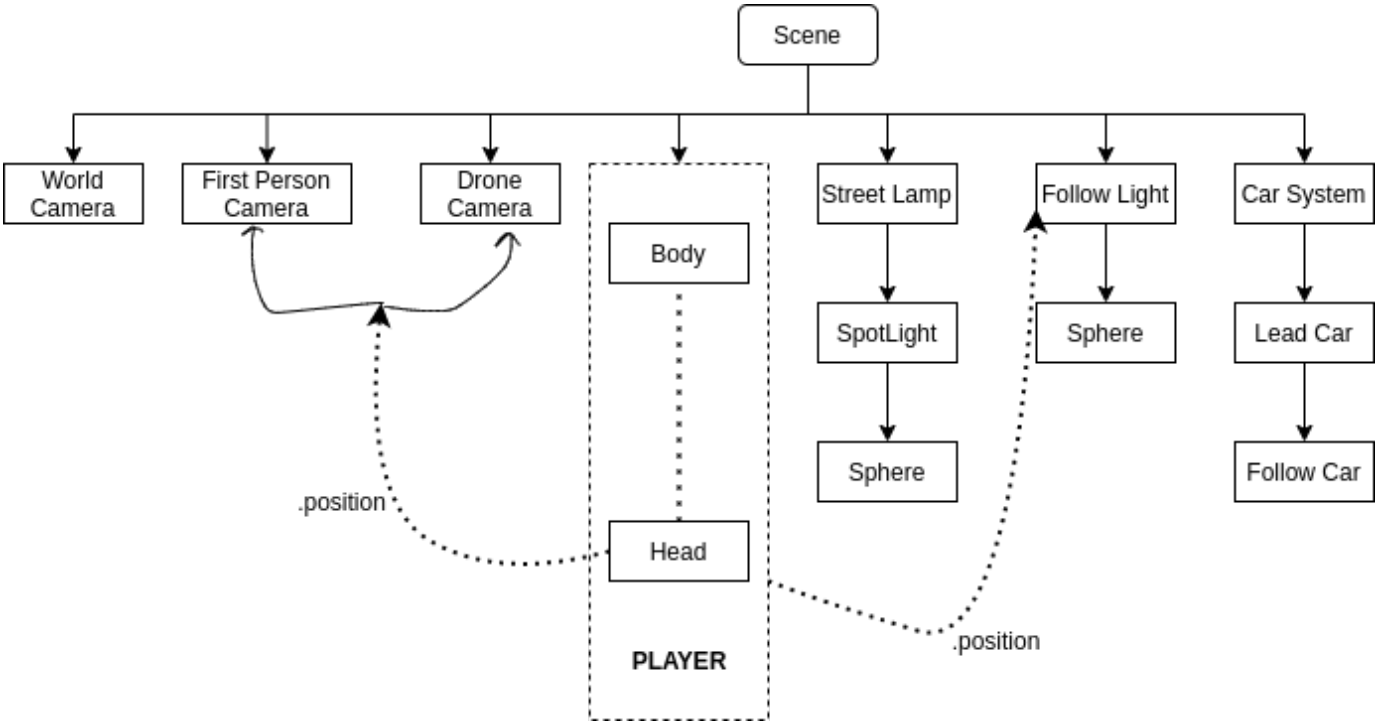

Fig. 1. Scene Graph for PLUNGE

`Head` descendent of the Player. As we mentioned before, here as well the positions have been defined with respect to the base and its final position has been calculated by application of matrices. So, for finding the camera's position, we traversed from the `Head` to the parent, multiplying the matrices in order to get the final position.

For the drone camera, to give it an effect of a drone that is following along a person, we decided to `lerp` the values to the new position instead of instant change. This addition of **noise makes the result realistic**.

C. Animations

Before rendering a scene, objects must be laid out in a scene. This defines spatial relationships between objects, including location and size. Animation refers to the temporal description of an object (i.e., how it moves and deforms over time. Now in our scene we needed a character that could move and play an animation sequence as he/she moves. An Animation Sequence is a single animation asset that can be played on a Skeletal Mesh(which is the base on which our mesh has been created). These contain keyframes that specify the position, rotation, and scale of a bone at a specific point in time. Using these Animation sequence and some nifty kinematic equations we were able to render an animated character into the scene.

Approach: So to achieve the above sequence we have a `requestAnimationFrame` function in JavaScript which calls a function repeatedly, in which we have out update and draw logic for three.js . Once, we call the `update` function it updates the position based on any control that is sent to the controller and then it renders the scene. So how does it move the object(kinematics) and how doe s it know which animations to play and when ? To achieve the animation sequence of when to trigger a particular animation, we use the **Finite State Machine**. The finite state machine essentailly disctates how and when we want an event to be triggered in the scene. For example: If a user has pressed w key, we would want to run the walk animation.
If the user has left all the key, then we would like to play the idle animation, and so on.

Along with this, we also need to create a **crossfade** between each transition so that the animation dont look choppy, and we are able to transition between them smoothly.

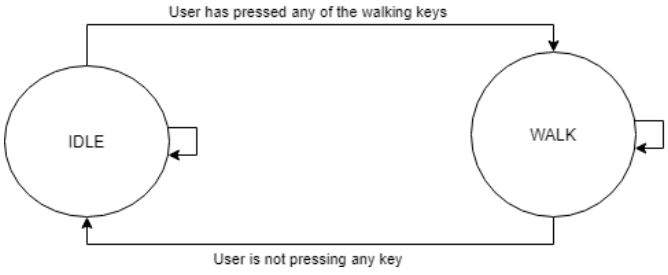

Fig. 2. Finite State Machine

D. Jumping and Collision

Approach: For enabling the Jumping facility, we utilised the periodic property of sine function. Since after every 2 radians, sine repeats itself, when we invoke the jump functionality by pressing the spacebar, we keep increasing the angle value that we are passing by a specific value(which determines how fast the player jumps) and when it cycles back and goes below a certain threshold, we stop the jumping sequence.

For checking collisions, even though there do exist complex methods using ray tracing, we divided to use a box-box intersection method for the purpose of this project. We used the Axis Aligned Bounding Boxes for each of the objects (Player and Cars for now). We continuously keep updating the transformation of the bounding boxes along with that of the object, to prevent changing the size of the bounding box as the object's orientation changes. For checking intersection we just use the `.intersects()` method of *Box3* object. Now, it is not like if the player hits an object it can not move in any direction. So to determine when it can move, we checked the angle between the vector from player's center to the object's center versus the vector from player's center to the point where the player is looking at. If this angle became more that 90, it meant that the player is free to move in that direction.

Problems: This was a major challenge because we had to change the bottom-most position of the object when it was on top of another object. So to enable this, I used a buffer of error, i.e. if the distance between the bottom of player bounding box was between a certain range with the top of object's bounding box, then we will change the base position of jumping to top of the box and set jumping variable to false. Now, this was the easy part and we were able to get flawless results. But the problem came when we had ensure that the object is able to walk off the object as well. One thing that we added was that, if the player is above an object it is free to move in any direction, so we removed that restriction. Then to enable the walking off the object functionality, if the player reaches a point when it is no longer in contact with the object, we check if the bottom-most point of the player is almost equal to the object's top most point then it must be free to fall, so we let it fall.

Errors In the current state, we are only able the object to get to climb an object when it intersects with it. Like the object can not jump from far and land on the object hoping to stay on top.

E. Leader Follow and Path Tracing

Approach: So for implementing a path tracing algorithm we made use of a **spline**. This spline was creates based on a set of track path inputs that are sent into the class. Now what about orientation ? So for orientation we compare the forward vector of our object with the nextMove vector (the direction we want to make the move towards). With the help of this we are able to get an angle between these two vectors which we use to gradually turn towards. This vector is also used in conjungtion with the angle orientation. As in if the angle is on the left of the forward vector , or on the right it finds a sign for the same , and is able to use that to find which direction we want to rotate in.

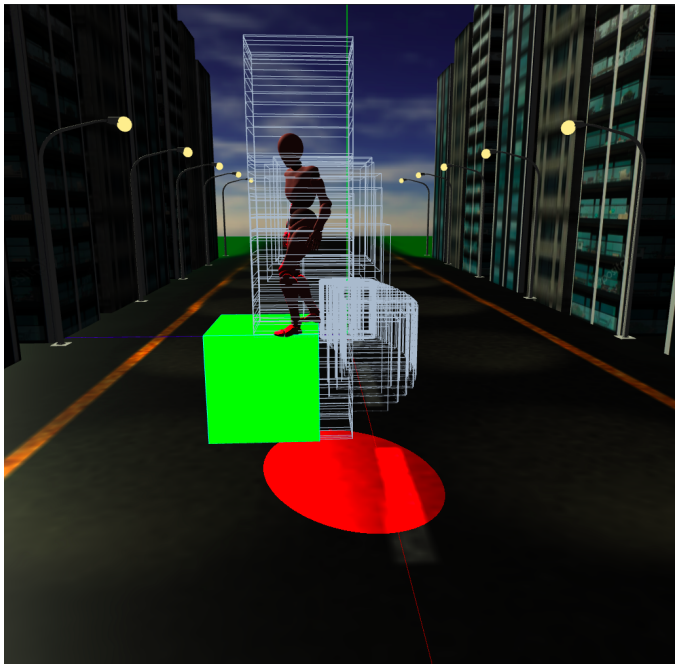

Fig. 3. Path of Bounding Box of character after jumping on a box

For the follow part, we generate a set of buffer points, by getting the parents position, once we have enough set of points, we generate splines for individual points to the car position and use that as a track for the object. Thus you could call this **dynamic path creation** for the child based on the parent's movement.

Since we have some margin of error while rotating, this noise all the more makes our model , seem more realistic.

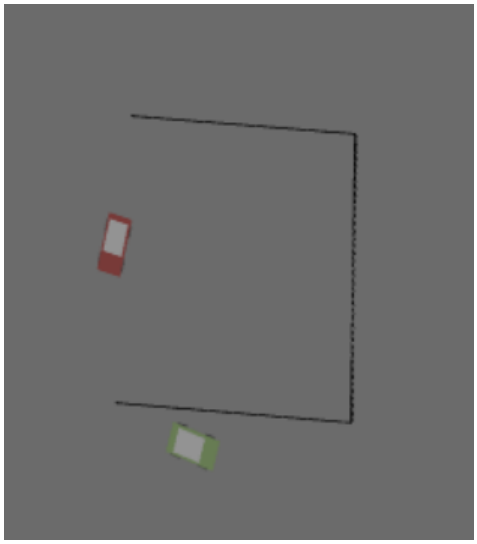

Fig. 4. Path Tracing and Follow Implementation

F. Textures

For textures we used blender to make the models and paint textures on them, in the building scene we have to use a building texture and paint it in the building using the texture as a stencil, we cant just use the texture on the building because it will distort it according to the UV mapping of the object.

UV mapping - it is the just the way the vertex of the object maps to a 2d plane, for every vertex there is a mapping to the 2d plane so the triangles made by the vertices also map to the plane, then we can map to the 3d object face the pixels of the 2d plane.

Different mapping results in different orientation of vertices, which in turn give different output of texture on the object.

Using different UV mapping technique with the same panted texture , we made a car model and painted it using blender with custom UV mapping, then we changed the UV mapping and keeping the texture constant, we see distortion of the texture on the object.

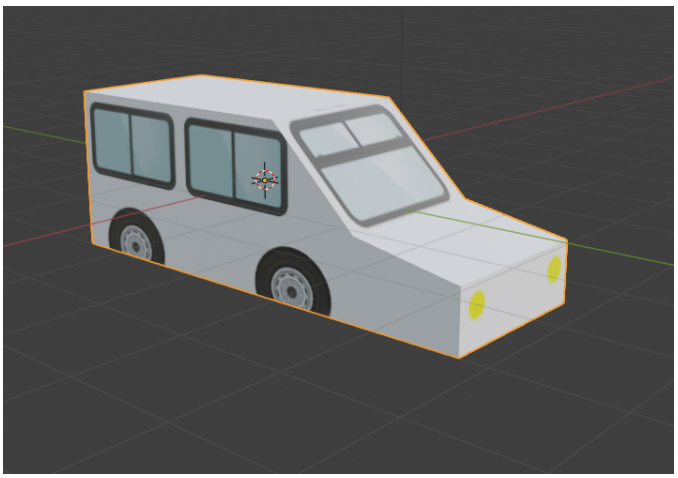

Fig. 5. custom mapping

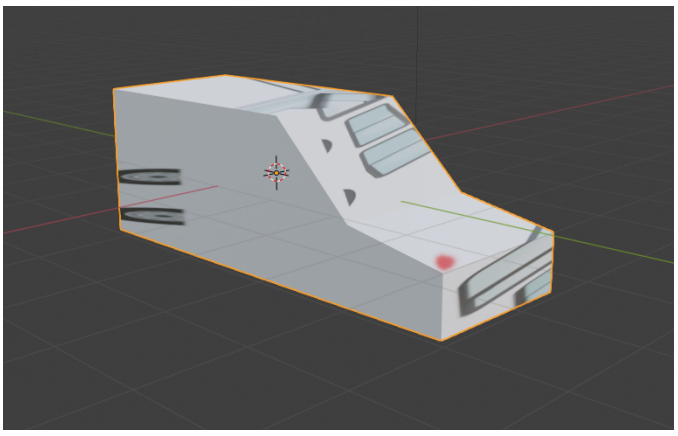

Fig. 6. cylinder mapping

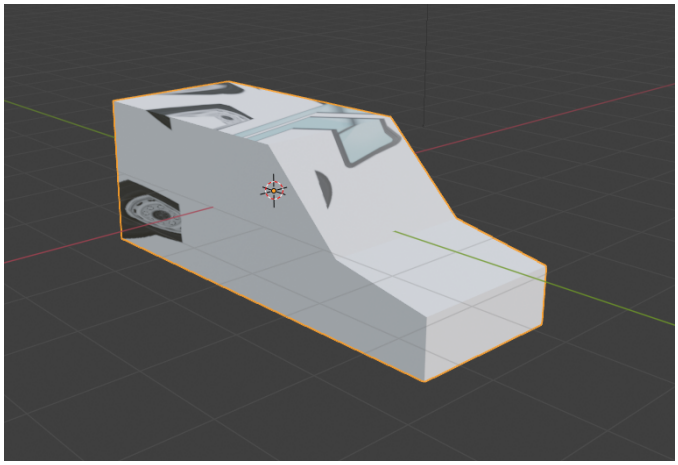

Fig. 7. sphere mapping

III. Conclusion

So, finally we were able to render an almost realistic scene which did follow the physics of collision to a large extent, were able to achieve lighting effects, utilise the different texture mappings to see the output and finally perform animations.

References