CG : Assignment 2 Report

Shreyansh Rai (IMT2020501) Vatsal Dhama (IMT2020029) Prakhar Rastogi (IMT2020052)

International Institute of Information Technology, Bangalore (IIITB)

Abstract

This assignment involves simulating a game played on a regular n-sided polygon-shaped playground, where m players are placed on different corners. One of the players is designated as the catcher and can move to any other corner in a straight line. Each corner can accommodate only one player, so if the catcher moves to a corner with an occupied player, that player must move to an unoccupied corner. The movement of players is illustrated through k steps taken on user command, and the game can be viewed from various camera angles in a graphics window.

Method

Shader Implementation

The Fragment Shader was not changed and uses the default implementation used in the Pacman assignment.

The Vertex Shader has significant changes to incorporate 3D viewing. The vertex shader program begins by defining attribute variables that will be used in the program. The aPosition

attribute represents the 3D position of a vertex in the scene, while aNormal represents the normal vector of the vertex. The program then declares uniform variables that are used to transform and project the 3D objects onto the 2D screen. uModelTransformMatrix is the transformation matrix that is used to translate, rotate, and scale the model in the scene. ViewMatrix is the view matrix that determines the camera's position and orientation in the scene, and ProjMatrix is the projection matrix that maps the 3D scene onto the 2D screen. The matrices are multiplied to apply the model matrix first, followed by the view and then projection matrix.

Platform (nGon)

The Platform class is a representation of a regular polygon-shaped playground in a simple game simulation. The class constructor takes in two arguments: n, the number of sides of the polygon, and color, the color of the platform. The class initializes several properties, including the side length, angle of each corner, radius, and normals of the polygon.

The generatePlatform method generates an equilateral polygon with n sides. It does so by first creating an empty array to hold the vertices of the polygon, then iterating through each corner of the polygon. For each corner, the method calculates the x and y coordinates using trigonometry and the radius of the polygon, adds the center point and the calculated x and y coordinates to the vertices array, and repeats for the next corner. The method then assigns the vertex coordinates to an array of only the vertices without the normals for easy use in collision detection.

The generate_random_vertex method generates a random index for a new corner position for the "catcher" player in the game. It uses the Math.random() method to generate a random number and the n property of the Platform object to ensure that the new index is within the bounds of the

polygon.

Camera Manipulation and The MouseControl class

Since this was one of the most key portions of the assignment I will go into substantial depth to explain this bit of the assignment. The MouseControls constructor takes the gl context as an input parameter and initializes various instance variables. The CurCam variable is an array that holds the current camera position, and the Cam variable is set to the same value initially. The deltax, deltay, and deltaz variables hold the changes in the mouse position along the x, y, and z axes, respectively. The thetax and thetay variables hold the angle of rotation around the x and y axes, respectively. The Held variable is set to zero initially and is used to determine whether the mouse button is held down. The AlterViewTo variable is a matrix that represents the current camera orientation. The Zoom variable holds the current zoom level, and the canvas variable holds the canvas element.

The canvas element is used to attach various event listeners such as mousedown, mouseup, mouseout, mousemove, and wheel. The mousedown event listener triggers the Press method of the MouseControls class, which sets the Held variable to one. The mouseup and mouseout event listeners trigger the Release method, which sets the Held variable to zero. The mousemove event listener triggers the Move method, which updates the deltax and deltay variables based on the mouse movement. Finally, the wheel event listener triggers the Scroll method, which updates the Zoom variable based on the scroll direction.

The alterView method of the MouseControls class is responsible for updating the camera position and orientation based on the mouse events. The method first updates the thetax and

thetay variables based on the changes in the mouse position. Then, it creates various matrices and vectors using the gl-matrix library. The quat2 library is used to perform quaternion rotations, which are more efficient than Eulerian rotations when dealing with 3D objects.

The quat2.rotateAroundAxis function is used to rotate the camera around the y-axis and x-axis based on the thetax and thetay variables, respectively. The resulting quaternion is stored in the q2 variable. Then, the mat4.fromQuat2 function is used to create a rotation matrix from the q2 quaternion, which is stored in the m1 variable.

Next, the current camera position vector is duplicated into another vector, and the vec3.transformMat4 function is used to transform the camera position vector by the rotation matrix. The resulting vector is negated along the x and y axes to position the camera correctly in the scene. Finally, the mat4.lookAt function is used to create the AlterViewTo matrix, which represents the new camera orientation.

The alterView method returns the AlterViewTo matrix, which can be used to render the scene from the new camera position and orientation.

In summary, this code uses quaternions to manipulate the camera orientation. The quat2.rotateAroundAxis function is used to perform the quaternion rotations, and the mat4.fromQuat2 function is used to create a rotation matrix from the resulting quaternion. The vec3.transformMat4 function is used to transform the camera position vector by the rotation matrix. Finally, the alterView() method returns the view matrix which can be used to transform the vertices of the 3D object in the scene to account for the new camera orientation and position.

The alterView() method is called every time the camera is moved or rotated in the Move() method, and also every time the zoom level is changed in the Scroll() method.

Object and 3D model loading

The "LoadedObject" constructor sets up several properties of the instance, such as "vertexPositions" and "vertexCoords" that respectively store the position data and 3D coordinates of each vertex in the model, "curr_vertex_index" that is initially -1, and "type" that is set to "model". "vertices" and "indices" are arrays containing the vertex and index data, respectively. The "convertmesh" method scales each vertex in "vertices" by a factor and adds it to "vertexCoords". The corresponding normal is also added to the "vertices" array, which is then converted to a Float32Array and stored in "vertexPositions". The "set_curr_vertex" method selects a specific vertex of the model for manipulation, and the "change_color" method updates the "color" property. The "LoadedObject" class has a "transform" property that is an instance of a class called "Transform", which may be used to transform the "LoadedObject" instance through scaling, rotation, or translation.

More on the Convert Mesh; The "convertmesh" method iterates through the "vertices" array and processes each vertex and its corresponding normal. The vertex is first scaled by a factor dependent on the model size in all three dimensions. This step may be necessary to make the model more visible, depending on its size and position. The scaled vertex is added to the "vertices" array, and its 3D coordinates are added to the "vertexCoords" array. The

corresponding normal is also added to the "vertices" array. Once all vertices have been processed, the "vertices" array is converted to a Float32Array and stored in "vertexPositions".

Renderer and Some deviations from the norm

The render() function is responsible for rendering the 3D scene in the WebGL canvas using the specified shader. It takes two parameters - scene and shader. I will explain the overall functionality and then in the last point is the key change that has been made. That is separate rendering based on primitive type.

The scene parameter is an object that contains all the 3D models and objects that need to be rendered. It has a property called primitives, which is an array of objects that represent the 3D models and objects in the scene. Each primitive object has properties such as vertexPositions, indices, color, and type.

The shader parameter is an instance of the Shader class, which contains the WebGL shader programs and methods to bind buffers, set uniform variables, and draw arrays or elements.

The first thing the render() function does is to set the projection and view matrices for the scene by calling setUniformMatrix4fv() method of the shader. These matrices are necessary for projecting the 3D objects onto the 2D canvas. The ProjMatrix is the projection matrix that maps the 3D coordinates to 2D screen coordinates, and the ViewMatrix is the view matrix that specifies the position and orientation of the camera.

After setting the matrices, the render() function iterates through each primitive in the scene.primitives array and performs the following steps:

1. Set the model transformation matrix for the primitive by calling setUniformMatrix4fv() method of the shader. This matrix specifies the position, rotation, and scale of the primitive in the 3D world.
2. Bind the index buffer for the primitive by calling bindIndexBuffer() method of the shader. The index buffer contains the indices of the vertices that form the triangles that make up the primitive.
3. Bind the vertex attributes buffer for the primitive by calling gl.bindBuffer() method of the shader. The vertex attributes buffer contains the vertex positions and normals for the primitive.
4. Populate the vertex attributes buffer data by calling gl.bufferData() method of the shader. This method copies the vertex positions and normals data from the primitive object to the buffer.
5. Enable the vertex attributes by calling gl.enableVertexAttribArray() method of the shader for both the position and normal attributes.
6. Set the uniform variable uColor for the primitive by calling setUniform4f() method of the shader. This sets the color of the primitive.
7. Draw the primitive by calling either drawArrays() or drawElements() method of the shader depending on the primitive type. If the primitive type is model, then drawElements() is called to draw the primitive using the index buffer. Otherwise, drawArrays() is called to draw the primitive using only the vertex positions**.**

Transforms

The constructor initializes instance variables such as "translate," "scale," "rotationAngle," and "rotationAxis" with default values using the "vec3" and "mat4" functions from the "gl-matrix" library. The instance variable "modelTransformMatrix" is also initialized with an identity matrix using the "mat4" function.

The "updateModelTransformMatrix" method takes a list of matrices as input, multiplies them together using the "mat4.multiply" function, and sets the result as the "modelTransformMatrix" instance variable. This method is called to update the model transformation matrix whenever any of the transform methods are called.

The "translateTransform," "rotateTransform," "rotateQuat," "scaleTransform," "translateRotateTransform," and "translateRotateScaleTransform" methods all create a new transformation matrix using various functions from the "mat4" library, such as "mat4.fromTranslation" and "mat4.fromScaling." These methods are used to generate matrices that represent specific transformations, such as translation or rotation, and are then multiplied together to create a final transformation matrix.

The "setRotate," "setScale," and "setTranslate" methods are used to directly set the rotation angle, scale, and translation instance variables, respectively. These methods are used to update the instance variables without creating a new transformation matrix.

Finally, the "reset" method sets the "modelTransformMatrix" instance variable to an identity matrix, effectively resetting the transformation. This method is used to clear any previous transformations and start from a clean slate.

For the ease of use of the person creating the game class, Additional functionality similar to previously implemented code in the First assignment is include although redundant, to ease transition into the newer code base.

Game

The Transform and Camera Controls classes are essential for implementing the game as they provide a convenient way to manage the position, orientation, and movement of the game objects. The Transform class allows the game objects to be positioned, rotated, and scaled in the 3D space, while the Camera Controls class manages the player's viewpoint and controls for the camera.

The object loader class, on the other hand, makes it easier to load and manipulate 3D models and other assets into the game. By using this class, the game can quickly load various objects and textures without having to handle the file loading and conversion process manually.

All of these classes work together to create a solid foundation for the game class implementation that follows.

The "colors_lst" attribute contains the list of unique colors of the players/cars, while the "vertex_lst" attribute contains the list of all the vertices of the polygon. The "can_go_lst" attribute is a boolean list that tells us if a particular vertex is accessible or not, i.e., if a car already exists there or not. The "obj_lst" attribute is a list of players/cars. The "curr_obj_index" attribute represents the player that is currently selected, while the "next_random_index" attribute represents the index to which the current player will be translated to. The "curr_color" attribute represents the color of the pixel that is clicked.

The "add_obj()" method is used to update the list of all cars/players. The "get_pixel_color()" method takes into input the mouse click coordinates and calculates the pixel coordinate using which it performs shaderpicking to find the color of the pixel that is clicked, setting the "curr_color" attribute.

The "get_random_index()" method takes into input the current vertex the player is at and generates another vertex that the player can be moved to. This method is recursive to ensure that the randomly generated vertex isn't the same as the current vertex.

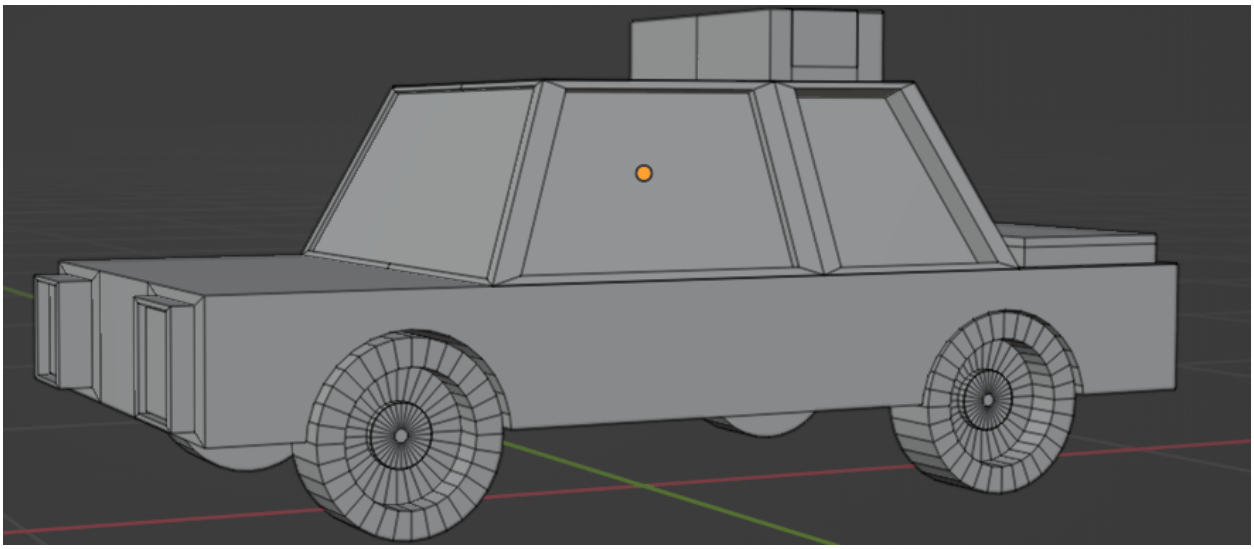
The "rotate_obj()" method rotates the current object by finding the angle between the vector joining the 2 points, i.e., the path of travel and the x-axis.

The "move()" method is responsible for the movement of the player. It first checks whether the vertex that was generated randomly has a player in it or not. This is done using the "can_go_lst". If the place is empty, then the player is directly translated to the new vertex, and the "can_go_lst" is updated, changing the player color back to the original from gray. In case there already exists a player at the generated vertex, the method first finds a new vertex where there is no player and translates the "already_there" object to this new vertex. Now since the generated vertex is empty, it translates the current player to the new vertex. Thus, collisions are also handled.

Blender

In blender we used a combination of mostly boolean differences and unions to get the outcome. So I shall describe the process of one of the models and a similar approach has been followed for the rest of the models. Below is the image of a car model that was made by us.The chassis is a

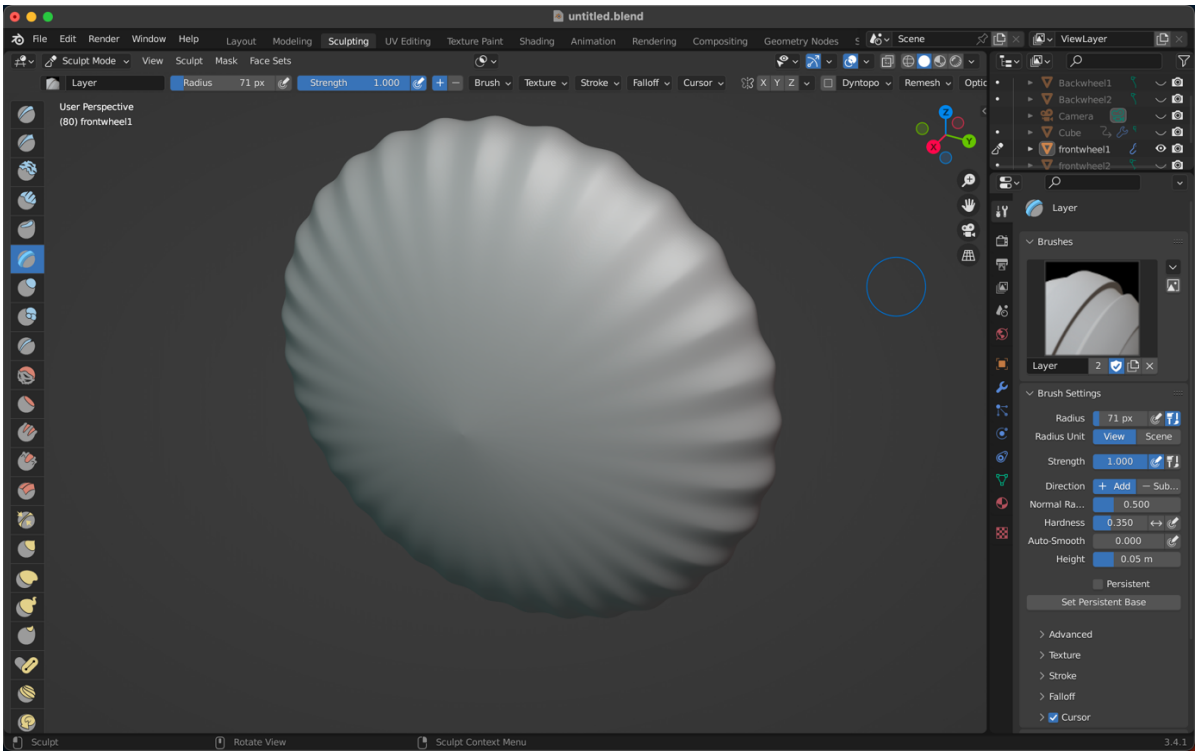
combination of cubes that were scaled and sheared in an appropriate way and were attached using a boolean union. The wheel was made using a Cylinder that was flattened and scaled and the unioned with the chassis and the cutouts for the wheel was made using boolean difference of a larger cylinder with the body of the car. The head lamps are also boolean unioned cube scaled appropriately. So extensive use of the boolean tool has been made in the building of all models. There was also extensive use of insetting new faces and extending them out to make it look aesthetic. An example would be the small stub on the wheel, it is an extended inset face.

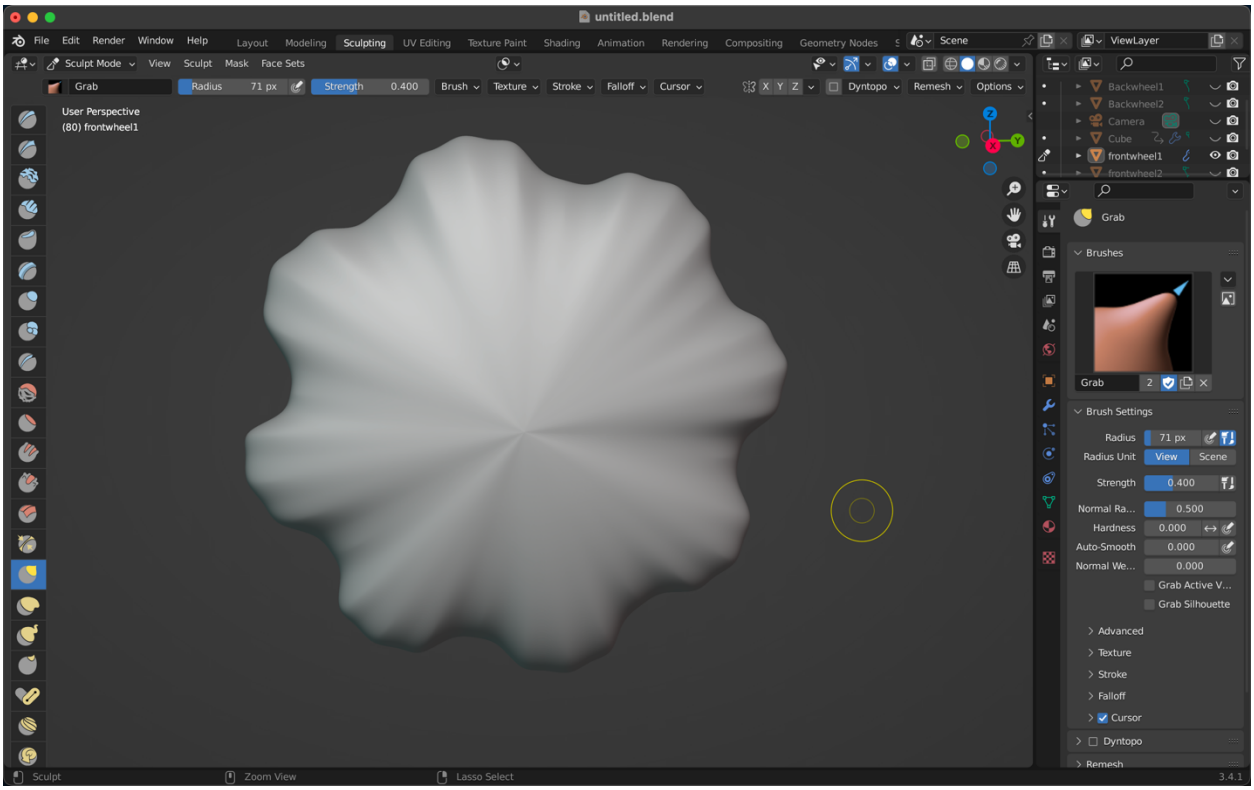


Sculpting

We tried Sculpting also. E.g we tried sculpting a cylinder in another model (using Grab and Layer modifiers as shown below.

But we were not able to render it as with sculpting the number of vertices increased drastically and WebGL was not able to load such large object. E.g. in the 2nd image below, the .OBJ file had approximately 30 lakh lines of code.


untitled.blend
File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Geometry Nodes
Scene
ViewLayer
Sculpt Mode View Sculpt Mask Face Sets
Layer
Radius 71 px
Strength 1.000
Brush Texture Stroke Falloff Cursor
Dyntopo Remesh
User Perspective
(80) frontwheel1
Backwheel1
Backwheel2
Camera
Cube
frontwheel1
Brushes
Brush Settings
Radius 71 px
Radius Unit View Scene
Strength 1.000
Direction Add Sub...
Normal Ra... 0.500
Hardness 0.350
Auto-Smooth 0.000
Height 0.05 m
Persistent
Set Persistent Base
Advanced
Texture
Stroke
Falloff
Cursor
Sculpt
Rotate View
Sculpt Context Menu
3.4.1

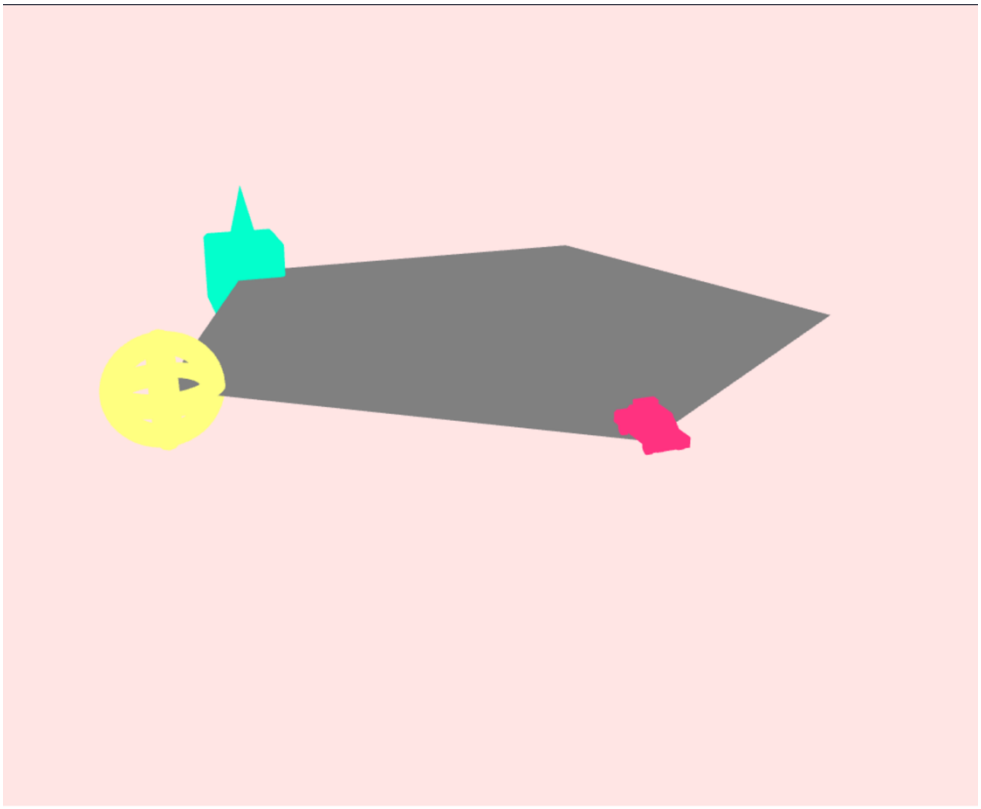

untitled.blend
File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Geometry Nodes
Scene
ViewLayer
Sculpt Mode View Sculpt Mask Face Sets
Grab
Radius 71 px
Strength 0.400
Brush Texture Stroke Falloff Cursor
Dyntopo Remesh Options
User Perspective
(80) frontwheel1
Backwheel1
Backwheel2
Camera
Cube
frontwheel1
Grab
Brushes
Brush Settings
Radius 71 px
Radius Unit View Scene
Strength 0.400
Normal Ra... 0.500
Hardness 0.000
Auto-Smooth 0.000
Normal We... 0.000
Grab Active V...
Grab Silhouette
Advanced
Texture
Stroke
Falloff
Cursor
Dyntopo
Remesh
Sculpt
Zoom View
Lasso Select
3.4.1

Controls

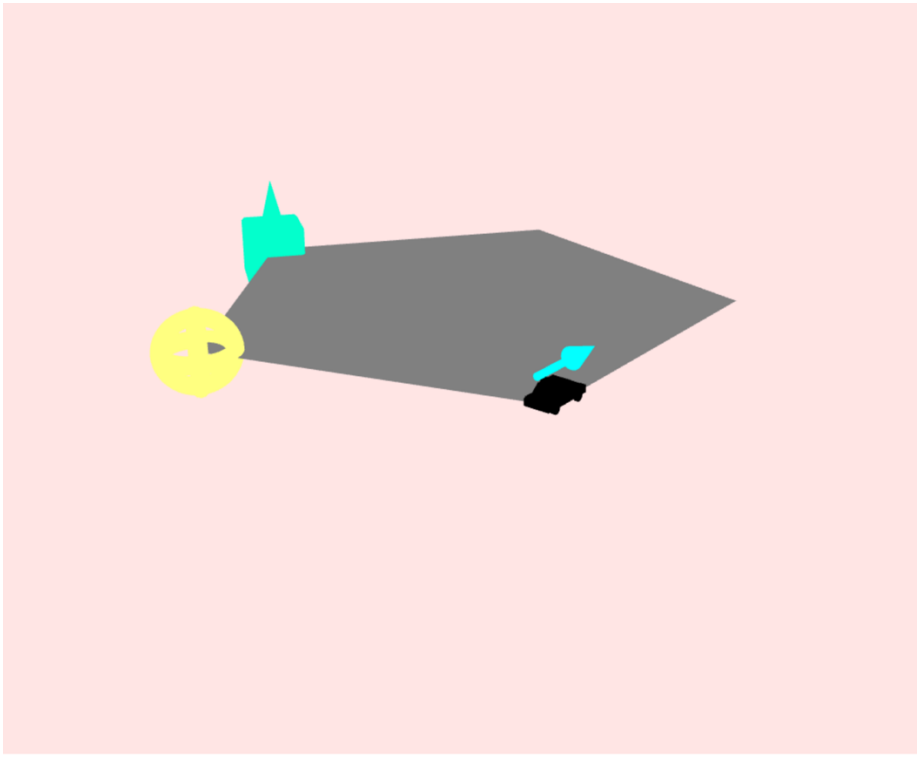
- Zoom : Scroll wheel
- View pan : left click and drag
- Enter Object pick mode : m
- Click on object and press enter to move it
- C to toggle god mode top down view

Images

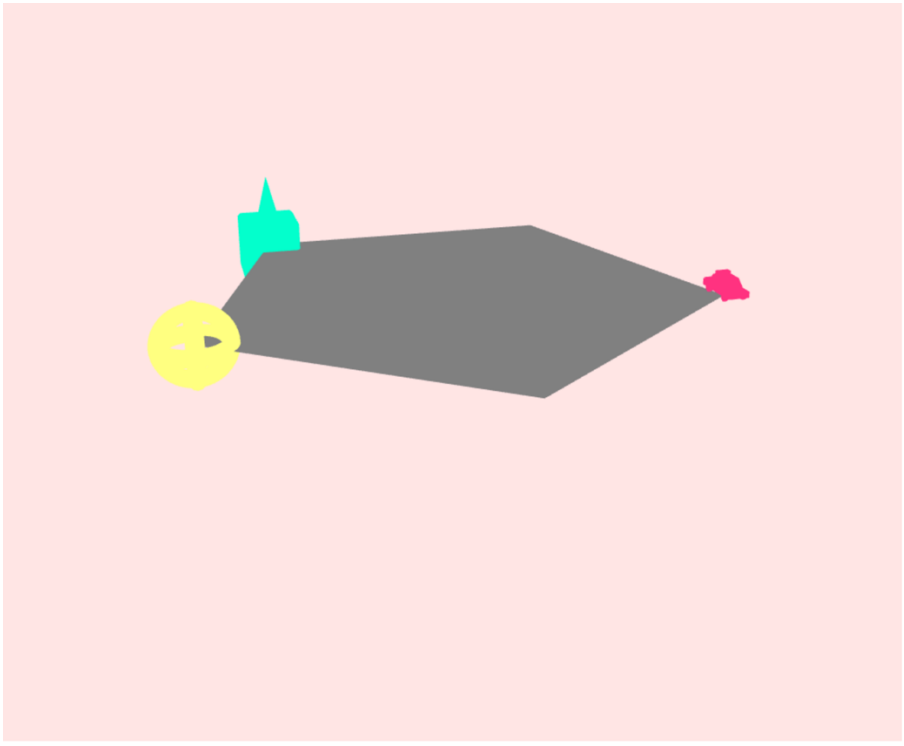
Starting position



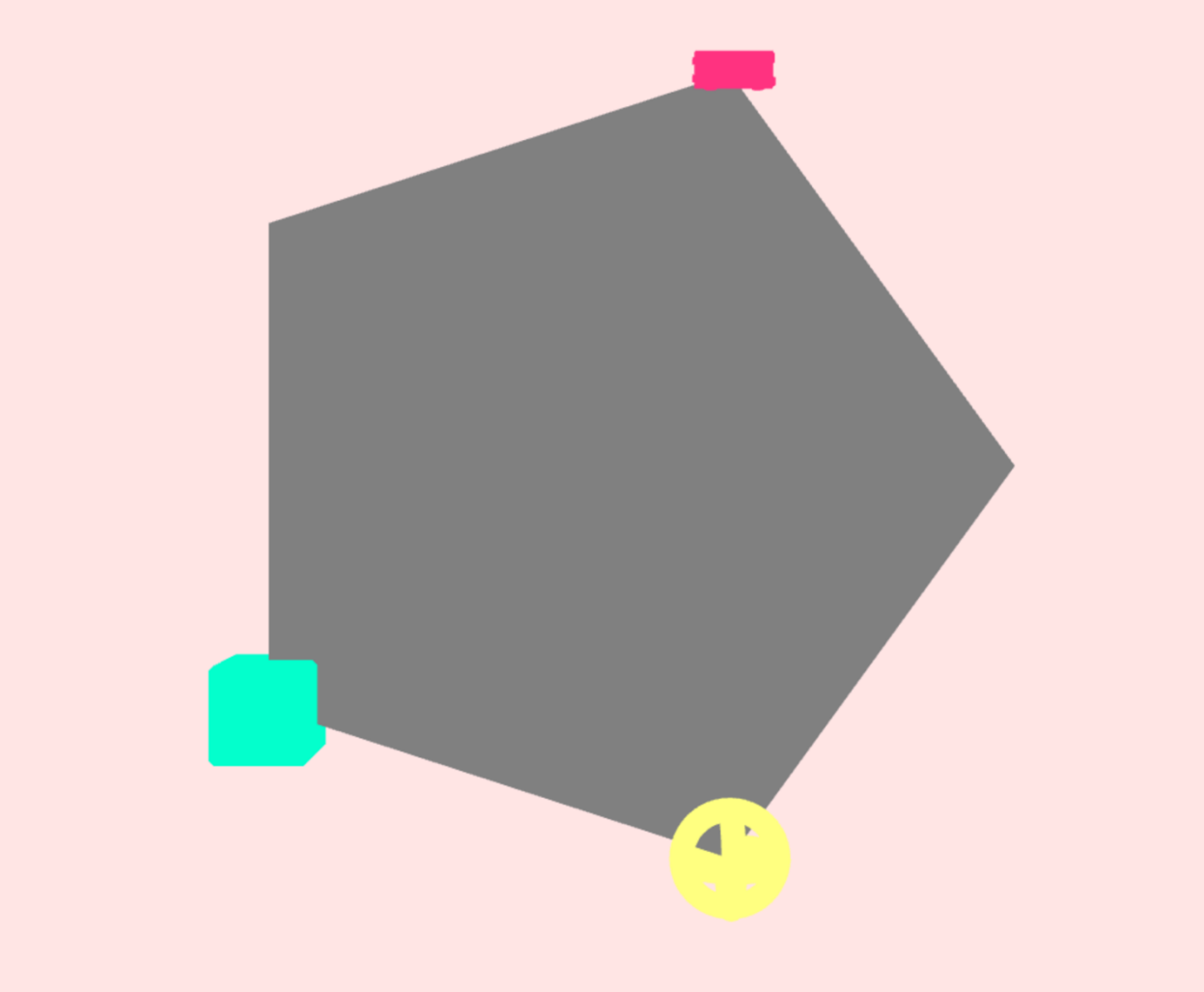
Selected position (with next vertex arrow)



New Position



God mode (top view) enabled by pressing the button C



Questions to be answered:

What were the primary changes in the use of WebGL in moving from a 2D world to a 3D world?

- There were completely new mouse view implementation that was done.
- Use of quaternions in the View changes
- Building an WebGL Object loader
- Use of rendering the mesh using vertex and index form. Which is a convenient way of defining the mesh.

How were the translate, scale and rotate matrices arranged and updated?

- Every operation is done with respect to origin
- For example all the movement operations are as follows:
    - Translate transform to origin
    - Rotate transform at origin
    - Translate to the destination coordinate
- For more information please refer the transformation section in the document

Describe at least 2 different approaches for using a mouse click to “select” an object in the scene

- First approach is that we could make all objects of different colors and store the mapping. Then we will fetch the color of the pixel on which we have clicked. We will check the mapping and find out which object was selected.
- Second approach we can use Raycasting. This method involves projecting a "ray" from the location of the camera towards the spot on the screen where the user clicked. The ray is then examined to determine whether it intersects with any objects present in the scene. If there is an intersection, then that particular object is chosen or highlighted.

Assume you had a large number of objects with the same original geometry (i.e. mesh, in this case). How would you organize the models so that you minimize the data stored?

- ⇨ To minimize the data stored for a large number of objects with the same original geometry, we can use instancing. Instancing is a technique that allows us to reuse the same geometry data for multiple objects, which reduces the amount of data that needs to be stored and transferred.
- ⇨ In our code all the objects that are added are first converted into a mesh format by our object loader, then they are assigned color and sent to the scene.
- ⇨ So, we can keep one mesh, and then reuse that same mesh

References

https://webglfundamentals.org/

How to pick an Object in WebGL part 1- ProgrammingTIL #121 WebGL tutorial video screencast 0064

Picking an object part 2 in 3d WebGL- ProgrammingTIL #122 WebGL tutorial video screencast 0065

Picking an object part 3 in WebGL - ProgrammingTIL #123 WebGL tutorial video screencast 0066