Tank Escape

Ryley Goodine, Zheyuan Xu

# Introduction

The goal of the game is for the player to move the blue cube tank to the end of the maze without touching the enemy cube tank.

Challenges faced with implementing the game were in the creation of the maze, how the player will be colliding with the maze walls, or the enemy cube tank and in initializing projectiles.

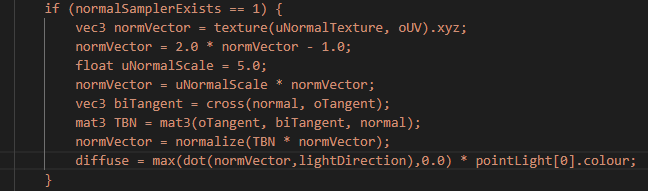
# Methods

## Playing Field / Terrain

图片包含 游戏机, 乐高

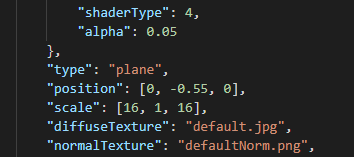
描述已自动生成Plane Object

* Summary of functionality
  + The plane object is used as the ground space for the scene. It does not interact with the objects through collisions fields as a physics engine was not included within the game. The plane texture is bump mapped and linked to the object inside the scene.json file.



* 图片包含 游戏机, 物体, 钟表, 球

  描述已自动生成Link to theory
  + The tangent vector is transformed and passed through in the vertex phase. Within the fragment phase, multiple calculations are performed to result in the TBN matrice.
  + A new normal vector is created from the specified texture and the linked UV values. The normal vector’s range is increased from 0-1 to -1-1.
  + The bitangent vector is calculated from the cross product of the normal vector and the tangent vector.
  + The TBN matrice is then calculated with input from the tangent, bitangent, and normal vectors.
  + The textured normal vector is then normalized with the TBN matrice, giving the new normal vector to be used within light calculations.
* Implementation details
  + Work from lab resources, particularly lab09 example03.js was used as the basis for the fragment shader code for the implementation of bump mapping within the scene.
  + The normal texture was provided as a default in Zach’s engine code.



## Main Player

A picture containing text

Description automatically generatedA picture containing blue

Description automatically generated

* Summary of functionality
  + The player object is a basic cube shape with colour calculated using the Bling-Phong model and diffuse texturing of the alien.jpg image.

vec3 textureColour = texture(uTexture, oUV).rgb;

diffuse = mix(diffuseVal, textureColour.rgb, 0.7);

* + The player is moved using the standard browser WASD key configuration.

this.player.translate(vec3.fromValues(+0.x, +0.0, +0.0)); // move left

this.player.translate(vec3.fromValues(-0.x, +0.0, +0.0)); // move right

this.player.translate(vec3.fromValues(+0.0, +0.0, +0.z)); // move forward

this.player.translate(vec3.fromValues(+0.0, +0.0, -0.z)); // move back

* Link to theory:
  + Movement of the player object on the scene is done through constant translation of the player objects current location.
  + The diffuse texture of the player object is calculated by mixing the uniform diffuseVal with the newly calculated texture value.
* Implementation details
  + Texture resources where used from the defaults provided in Zach’s engine, and the texture colour equation was taken from assignment06.

## Interacting Objects

* Walls

A picture containing text, outdoor, building material, blue

Description automatically generated

* + Summary of functionality
    - Collision detection for the player object, it prevents the player from moving in the direction of the wall.
    - Each wall object has a sphere collider connected to it that specifies the object name, and the affected radius.

createSphereCollider(object, radius, onCollide = null) {

        object.finish=0;

        object.stop=vec3.fromValues(0,0,0);

        object.collider = {

            type: "SPHERE",

            radius: radius,

            onCollide: onCollide ? onCollide : (otherObject) => {

                //console.log(`Collided with ${otherObject.name}`);

                //object.stop=vec3.fromValues(1,1,1);

            }

        };

        this.collidableObjects.push(object);

    }

const wallObject1 = getObject(this.state, "myWall5");

this.createSphereCollider(wallObject5, 0.4);

* + - With the stop vector variable declared during the creation of the collider, the player object will stop movement in each direction of WASD.
    - Such as for the keypress of “a”, as long as the player’s stop vector value in the x-coordinate is less then or equal to 0, the player will be able to move.

if (this.player.stop[0] <= 0) {

this.player.translate(vec3.fromValues(0.25, 0, 0));

* + Link to theory
    - Sphere collision involves testing if the distance between the two centers is larger then the sum of the two radius.

(distance < (object.collider.radius + otherObject.collider.radius))

* + Implementation details
    - Using the sphere collider code was taken from lab10, and player movement was modified
    - Using Zacks game engine to scale
    - Blending diffuse light with texture colour
    - The entire wall is achieved by placing multiple cubes in line and adding spherical colliders to each cube.
* Finish line

A picture containing text

Description automatically generated

* + Summary of functionality
    - Collision detection for the player object, it indicates that the player has reached the end of the game or has hit the enemy player. Sphere collision is used, and the values are stored within “finish” variable. When the player object collides with the object name “finish”, the player’s finish value is set to 1. From movement of the player, the player is able to traverse through the finish object

if ((otherObject.name === "finish") && (distance < (object.collider.radius + otherObject.collider.radius))) {

this.player.finish = 1;

return;

}

* + - Within the onUpdate function, the code constantly checks if the player’s finish value has changed into 1 or 2.

if (this.player.finish === 1) {

gameState.innerHTML = "YOU WON!";

gameState.style.color = "green";

}

* + - If the player has moved within distance of the “finish” object, then the HTML text will change its display to indicate the game is over and that the player has won.
  + Link to theory
    - Sphere collision is used to check if the player object has moved within the distance of the “finish” object.
    - Visibility and Transparency
  + Implementation details
    - The code from lab10 for the collision detection of the other objects was modified to check if the object has a particular name, and then perform a specific action.
    - Sorting objects based on the depth and adding alpha value to the rendering
    - To avoid transparent objects occluding other objects, we first render opaque objects with z-buffering on, then transparent objects with the z-write component of z-buffering off gl.depthMask(false).

## Non-Player Character

A picture containing underpants

Description automatically generated A picture containing furniture, seat, chair

Description automatically generated

* Summary of functionality
  + The enemy red tank, it moves in a fixed loop around its position. If the player collides with it, they will lose the game.

if ((otherObject.name === "myNPC") && (distance < (object.collider.radius + otherObject.collider.radius))) {

this.player.finish = 2;

vec3.subtract(object.stop, otherMatrix, objectMatrix);

return;

}

* + How it links inside the onUpdate function to constantly check if the player has collided with the NPC

if (this.player.finish === 2) {

gameState.innerHTML = "YOU LOST!";

gameState.style.color = "red";

}

* + Movement of the NPC object is also done inside the onUpdate function with the use of a counter that increments with deltaTime.

if(this.n === 150){

            npcObject.translate(vec3.fromValues(0, 0, 1));

        }else if(this.n === 300){

            npcObject.translate(vec3.fromValues(0, 0, -1));

        }else if(this.n === 450){

            npcObject.translate(vec3.fromValues(0, 0, -1));

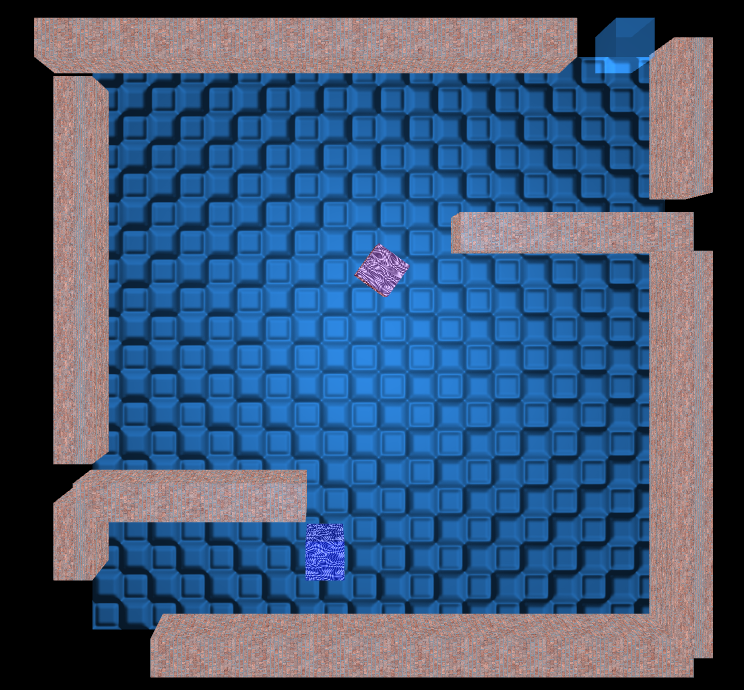
        }else if(this.n === 600){

            npcObject.translate(vec3.fromValues(0, 0, 1));

}

* + The object is translated to the position in the scene based on the counter value to create a loop of movement.
* Link to theory
  + Creation of the model with its texture, and movement around the scene at fixed locations.
  + Counter iteration with if/ else if statements checking if the counter equals the specified value. Collision detection through the sphere collider, similar to the walls and player object. An additional check during collision is made for if the player collides with the NPC, in which case they lose the game.
* Implementation details
  + Texture image and colour values retrieved from the scene.js file. Its then linked into the fragment and vertice shader where the Blinn-Phong model equation will be used to calculate colour. Texture colour is calculated based on the shaderType value, if it is 3 then the diffuse texture will be used, if it is 4 then the normal texture with bump mapping.
  + Through our own custom method to count the time of each frame, and then if-else judgment to control the NPC in a fixed time to come back and cycle the blocking movement.
  + Adding a spherical collider.

## Change of View

 A picture containing blue, bedclothes

Description automatically generated

* Summary of functionality
  + Two types of views included; top-down and first-person.
* state.cameraKey = 0;
* state.camera = [{
* name: "topCamera",
* position: vec3.fromValues(1.0, 7.0, 1.0),
* front: vec3.fromValues(1.0, 0.0, 1.0),
* up: vec3.fromValues(0.0, 0.0, 1.0),
* }, {
* name: "povCamera",
* position: vec3.fromValues(3.75, 0.75, -3.75),
* front: vec3.fromValues(0.0, 0.0, 2.0),
* up: vec3.fromValues(0.0, 1.0, 0.0),
* }];
  + Two camera’s are created within the main.js file.
  + With the cameraKey variable in state, it can retrieve the desired camera, and allows for additional camera positions to be added with minimal code required for correct changes in the view matrix.
* // View Matrix & Camera ....
* let viewMatrix = mat4.create();
* mat4.lookAt(
* viewMatrix,
* state.camera[state.cameraKey].position,
* state.camera[state.cameraKey].front,
* state.camera[state.cameraKey].up,
* );
* gl.uniformMatrix4fv(object.programInfo.uniformLocations.view, false, viewMatrix);
* gl.uniform3fv(object.programInfo.uniformLocations.cameraPosition, state.camera[state.cameraKey].position);
* state.viewMatrix = viewMatrix;
  + Within the game scene, the [f] key is linked with changing the selected camera to the opposite one on each key press.

case "f":

document.getElementById("f").style.color = this.keyDownColor;

if (this.state.cameraKey == 0) {

this.state.cameraKey = 1;

} else {

this.state.cameraKey = 0;

}

break;

* + The first person camera is rotated along the front value using the [q] and [e] keys.

case "q":

document.getElementById("q").style.color = this.keyDownColor;

vec3.add(this.state.camera[1].front, this.state.camera[1].front, vec3.fromValues(0.25, 0, 0));

break;

case "e":

document.getElementById("e").style.color = this.keyDownColor;

vec3.add(this.state.camera[1].front, this.state.camera[1].front, vec3.fromValues(-0.25, 0, 0));

break;

* Link to theory
  + Use of the view matrix by calculating the look-at value through the camera’s position, front, and up vectors.
* Implementation details
  + Two cameras that are initialized within the main javascript file outside of the game.
  + Top down view and first person view are both initially set.
  + First person view follows the movement of the player within the scene.
  + Within drawscene function, camera positioning is indicated with a list index. This affords more flexibility within the game, and allows more camera views without having to include additional code.

## Additional Functionality

* Adding sound from projectiles

case " ":

document.getElementById("space").style.color = this.keyDownColor;

this.createBullet();

projectileNoise.play();

break;

* + Summary of functionality
    - Sound file is loaded from the HTML with the audio tag and included with an id.
    - Within the game.js file inside the onPlay function, the specific id is retrieved and linked to a constant variable.
    - The sound is sped up to the maximum amount to allow for multiple firings.
    - Hitting the space key, which creates the projectile, will now also play the sound
* Adding basic timer

document.getElementById("timer").innerHTML = new Date(Date.now()).toISOString();

* + Summary of functionality
    - A built-in javascript world-timer without modification
    - It links to an id within the HTML file, and is constantly updated within the onUpdate function in game.js.
* Adding keymapping to html
* <table>
* <tr style="text-align:center;">
* <td id="q">Q<br>rotate left</br></td>
* <td id="w">W<br>forward</br></td>
* <td id="e">E<br>rotate right</br></td>
* <!--<td id="r">R<br>reset</br></td>-->
* </tr>
* <tr style="text-align:center;">
* <td id="a">A<br>left</br></td>
* <td id="s">S<br>back</br></td>
* <td id="d">D<br>right</br></td>
* <td id="f">F<br>swap view</br></td>
* </tr>
* </table>
  + Summary of functionality
    - The indicated keys are linked to their respective HTML tag within both the keyup and keydown listener. The HTML document links to the corresponding id, with its contents formatted in a table.
    - Two event listeners are added; keydown and keyup which change the colour of the text to a defined colour within the game.js constructor.
* this.keyUpColor = "white";
* this.keyDownColor = "green";
* document.getElementById("f").style.color = this.keyDownColor;
* document.getElementById("f").style.color = this.keyUpColor;
* Music
* const songName = document.getElementById("currentMusic");
* songName.innerHTML = this.musicPlayListNames[this.musicIndex];
  + Summary of functionality
    - Audio is added inside the HTML file using the audio tag.
    - Soundtrack’s are stored within a list, and the current index is stored within the game variable. The name of the song is linked inside a separate list and linked to an HTML header tag to display output to the user.
* case "m":
* document.getElementById("m").style.color = this.keyDownColor;
* if (this.musicPlayList[this.musicIndex].paused) {
* this.musicPlayList[this.musicIndex].play();
* } else {
* this.musicPlayList[this.musicIndex].pause();
* document.getElementById("m").style.color = this.keyDownColor;
* }
* break;
  + - Pressing the [m] key plays and pauses the song, and while the song is playing the keyDownColor will stay active.
* case "ArrowRight":
* document.getElementById("ArrowRight").style.color = this.keyDownColor;
* this.musicPlayList[this.musicIndex].pause();
* this.musicPlayList[this.musicIndex].currentTime = 0;
* if (this.musicIndex === this.musicPlayList.length-1) {
* this.musicIndex=0;
* } else {
* this.musicIndex++;
* }
* //console.log("right key pressed");
* songName.innerHTML = this.musicPlayListNames[this.musicIndex];
* break;
  + - Using the arrow keys cycles through the song selection, stopping the current song if it already played
  + Link to theory
  + Implementation details
* Changing scene composition based on played song
* customMusicGameChange() {
* switch(this.musicIndex) {
* // Personal Jesus
* case 0: {
* this.pointLightCycle = 0;
* this.changeLights = true;
* break;
* }
* // Won't Get Fooled Again
* case 1: {
* this.changeLights = false;
* this.pointLightColours = [0.1, 0.3, 0.5];
* this.state.pointLights[0].colour[0] = this.pointLightColours[0];
* this.state.pointLights[0].colour[1] = this.pointLightColours[1];
* this.state.pointLights[0].colour[2] = this.pointLightColours[2];
* break;
* }
* // Juke Box Hero
* case 2: {
* this.changeLights = false;
* this.pointLightColours = [1, 1, 1];
* this.state.pointLights[0].colour[0] = this.pointLightColours[0];
* this.state.pointLights[0].colour[1] = this.pointLightColours[1];
* this.state.pointLights[0].colour[2] = this.pointLightColours[2];
* this.bulletColor[0] = [1, 1, 1];
* this.bulletColor[1] = [1, 1, 1];
* this.bulletColor[2] = [1, 1, 1];
* this.bulletColor[3] = 1;
* this.bulletColor[4] = 50;
* break;
* }
* // Back In Black
* case 3: {
* this.changeLights = false;
* this.pointLightColours = [0, 0, 0];
* this.state.pointLights[0].colour[0] = this.pointLightColours[0];
* this.state.pointLights[0].colour[1] = this.pointLightColours[1];
* this.state.pointLights[0].colour[2] = this.pointLightColours[2];
* this.bulletColor[0] = [1, 1, 1];
* this.bulletColor[1] = [1, 1, 1];
* this.bulletColor[2] = [1, 1, 1];
* this.bulletColor[3] = 1;
* this.bulletColor[4] = 50;
* break;
* }
* // Refugee
* case 4: {
* this.changeLights = false;
* this.pointLightColours = [0.1, 0.55, 0.1];
* this.state.pointLights[0].colour[0] = this.pointLightColours[0];
* this.state.pointLights[0].colour[1] = this.pointLightColours[1];
* this.state.pointLights[0].colour[2] = this.pointLightColours[2];
* this.bulletColor[0] = [0, 0.5, 0];
* this.bulletColor[1] = [0, 1, 0];
* this.bulletColor[2] = [0, 0.5, 0];
* this.bulletColor[3] = 1;
* this.bulletColor[4] = 50;
* break;
* }
* // Spaceship Superstar
* case 5: {
* this.changeLights = true;
* this.pointLightCycle = 4;
* this.pointLightMax =  0.5;
* this.pointLightMin = -0.5;
* this.bulletColor[0] = randomVec3(0,1);
* this.bulletColor[1] = randomVec3(0,1);
* this.bulletColor[2] = randomVec3(0,1);
* this.bulletColor[3] = 1;
* this.bulletColor[4] = 50;
* break;
* }
* default: {
* this.changeLights = true;
* this.bulletColor[0] = [0.25, 0.75, 0.25];
* this.bulletColor[1] = [0.00, 1.00, 0.00];
* this.bulletColor[2] = [0.25, 1.00, 0.25];
* this.bulletColor[3] = 0.5;
* this.bulletColor[4] = 25;
* this.state.pointLights[0].colour[0] = this.pointLightColours[0];
* this.state.pointLights[0].colour[1] = this.pointLightColours[1];
* this.state.pointLights[0].colour[2] = this.pointLightColours[2];
* break;
* }
* }
* }
  + Each song affects the global light colour, and if it cycles as well as the bullet colours.
* Changing the colour of the main point light

this.changeLights = true;

this.pointLightCycle=0;

this.pointLightColours=[0, 0, 0];

this.pointLightC=[(1/255),(1/255),(1/255)];

this.pointLightMax =  0.5;

this.pointLightMin = -0.5;

pointLightCycleRed() {

if (this.pointLightCycle<2) {

this.pointLightColours[0]+=(this.pointLightC[0]);

if (this.pointLightColours[0]>=this.pointLightMax){

this.pointLightC[0]= (-this.pointLightC[0]);

} else if (this.pointLightColours[0]<this.pointLightMin){

this.pointLightC[0]= (-this.pointLightC[0]);

this.pointLightCycle++;

}

}

}

pointLightCycleGreen() {

if (this.pointLightCycle>=1) {

this.pointLightColours[1]+=(this.pointLightC[1]);

if (this.pointLightColours[1]>=this.pointLightMax){

this.pointLightC[1]= (-this.pointLightC[1]);

} else if (this.pointLightColours[1]<this.pointLightMin){

this.pointLightC[1]= (-this.pointLightC[1]);

this.pointLightCycle++;

}

}

}

pointLightCycleBlue() {

if (this.pointLightCycle>=2) {

this.pointLightColours[2]+=(this.pointLightC[2]);

if (this.pointLightColours[2]>=this.pointLightMax){

this.pointLightC[2]= (-this.pointLightC[2]);

} else if (this.pointLightColours[2]<this.pointLightMin){

this.pointLightC[2]= (-this.pointLightC[2]);

this.pointLightCycle++;

}

}

}

* + Summary of functionality
    - The global light is run through a progressive updater through each RGB colour.
    - The starting light colour is set to black, and the starting counter is set to zero. The pointlight is added with increments if 1/255, and the maximum and minimum ranges are set. Within the functions, based on what iteration the counter is on, the selected RGB colour will increase itself until it reaches the fixed maximum and then move in the opposite direction. After it has reached the minimum value, the counter will increase

if (this.changeLights === true) {

            if (this.pointLightCycle === 3) {

                this.pointLightCycle = 0;

            }

            this.pointLightCycleRed();

            this.pointLightCycleGreen();

            this.pointLightCycleBlue();

this.state.pointLights[0].colour[0] = this.pointLightColours[0];

this.state.pointLights[0].colour[1] = this.pointLightColours[1];

this.state.pointLights[0].colour[2] = this.pointLightColours[2];

}

* + - Within the onUpdate function, it constantly checks if the changeLight value is equal to true, then calls the three colour change functions. When the counter reaches 3 it resets itself back to 0. After each function call, the state pointLight colour is updated with the new pointLight values.

# Analysis and discussion

* Summarize what we did

We learned through experiment 10 many techniques for using zach's engine including and not limited to apply a diffuse texture, object movement rotation, scene construction, collision volume calculation, etc.

* What we learned from this project

WebGL 3D graphics processing, including position determination, light position processing and light color transformation, object movement rotation and collision processing, object texture, etc.

* What worked and what didn’t (why)

We tried to design a new rectangular collision detection, but for time reasons we did not design a suitable rectangular collision detection, so we chose to follow the spherical collision one, using multiple cubes to form a whole wall, to achieve the same effect of preventing the tank from moving through the wall.

* What would we do to continue/improve

Projectile collision effects and collision sound effects

Better rectangular collision determination

Credits:

Zheyuan XU: Game concept discussion, scene building, protagonist movement as well as camera rotation, Implement bump mapping, so that the plane to achieve the effect of 3d, NPC as well as wall collision adding testing, NPC movement, testing, and short video demo recording and ppt writing.

Ryley Goodine: Game concept discussion, scene building, player movement and initialization of two camera views, fragment shader blinn-phong equation and vertex shader normal mapping code addition. Projectile creation of bullets including its destruction after defined distance from player or number of existing bullets in scene reaches arbitrary amount. Cyclical global light change through iterative addition and subtraction of fixed amount. Implementation of HTML additions including music, sound form bullet activation, dynamic updates in html table on keypresses­