"Road Runner"

Jihae Han
CG Report

I: Overview

Road Runner is a racing game set in a surreal world. The player drives a motorcycle along a topsy-turvy route that travels in between painterly hills and strange floating pavilions. The game is played from third person view, but other camera views - side view and top-down view - can also be triggered. The player aims to get the highest score possible by picking up floating spheres and dodging harmful spikes along the route. Beware, as the player only has three lives to accomplish this task.

A prototype mockup can be seen in *Figure 1* below. The report will describe the development of the project under the following sections: II Route and Camera, III Objects, Meshes, and Lighting and IV: HUD, Gameplay, and Advanced Rendering. The final section V will attribute assets and works cited, and include a discussion of the project.

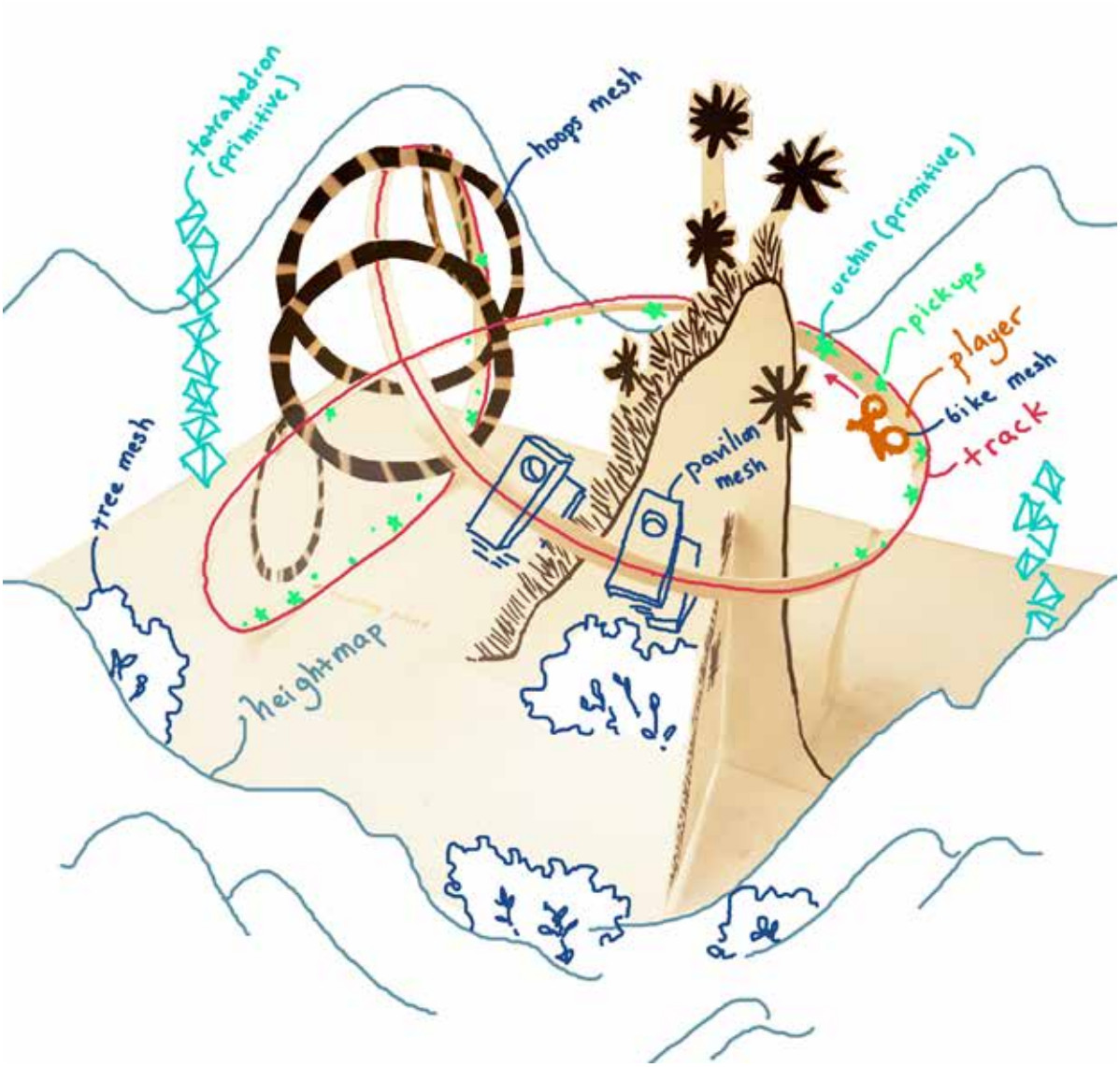
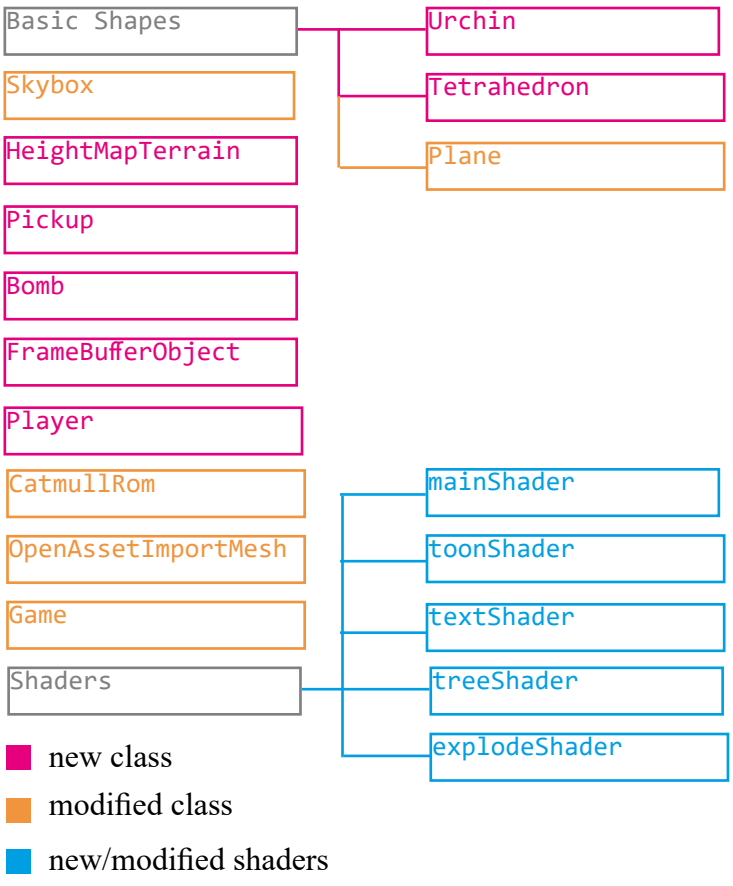


Figure 1: Prototype

Figure 2: Keyboard Controls

Player Controls	W	move player forward
	A	move player left
	S	move player backward
	D	move player right
Camera Controls	F	toggle freeview (toggle again for game restart)
	F1	3rd person camera
	F2	side view camera
	F3	top-down view camera
Scener Rendering	TAB	toggle 'dark' mode
	M	toggle real-time minimap
Colour Filters	0	no filters
	1	black and white mode
	2	colour shift
	3	extra vibrant
	4	rainbow colour mode
	5	inverted colours mode
General	SPACE	enter into gameplay/instructions state
	ALT	pause game

Figure 3: Class Overview



The main keyboard controls in the game are listed above in *Figure 2*.

An overarching view of the classes in the game are listed on the left, *Figure 3*, with emphasis on the classes that were either added or modified into the original OpenGL template. The shaders listed have been either added or extensively modified from the original template.

All requirements have been met in all sections.

II: Route and Camera

All requirements have been implemented for this section.

A 3D non-linear path has been generate programmatically using the CatmullRom class and implemented in the game scene. Triangular primitives are added on the centreline to generate a textured surface that represents the tracing track. The screenshots on the right, *Figure 4*, portrays a semi-transparent textured primitive. Different lighting settings have been set for the track texture, so that 'day mode' renders the track with higher transparency than 'night mode'. This was an aesthetic choice to make daytime more cheery and light, while nighttime darker and denser.

Flat tetrahedrons demarcate the edge of the route like a strange spinal-cord , as seen in *Figure 5*. As the track is a 3D spline, the tetrahedrons too need to be rotated in three dimensions. Helper functions `Game::RotationBetweenVectors` and `Game::LookAt` have been generated to apply quaternion mathematics on matrix rotation. The necessary rotations are stored in a vector of quaternnions upon initialization, to minimize the computational pressure in real-time.

CatmullRom.cpp (rendering track walls)

```
//initialize offset track demarcators
for (int i = 0; i < 100; i++) {
   int j = i * 4;
   wall_lefttrack.push_back(
        m_pCatmullRom->GetLeftOffsetPoints()[j]);
   wall_lefttrackrot.push_back(
        glm::toMat4(Game::LookAt(
        m_pCatmullRom->GetUpPoints()[j],
        -m_pCatmullRom->GetOffsetPoints()[j])));
   //and similarly for right offset walls
}
```

```
//Render offset track demarcators
for (int i = 0; i < 100; i++){
    modelViewMatrixStack.Push();
    modelViewMatrixStack.Translate(
        wall_lefttrack[i]);
    modelViewMatrixStack.RotateQuat(
        wall_lefttrackrot[i]);
    //setup shader program
    m_pTrackWall->Render();
    modelViewMatrixStack.Pop();
}
```

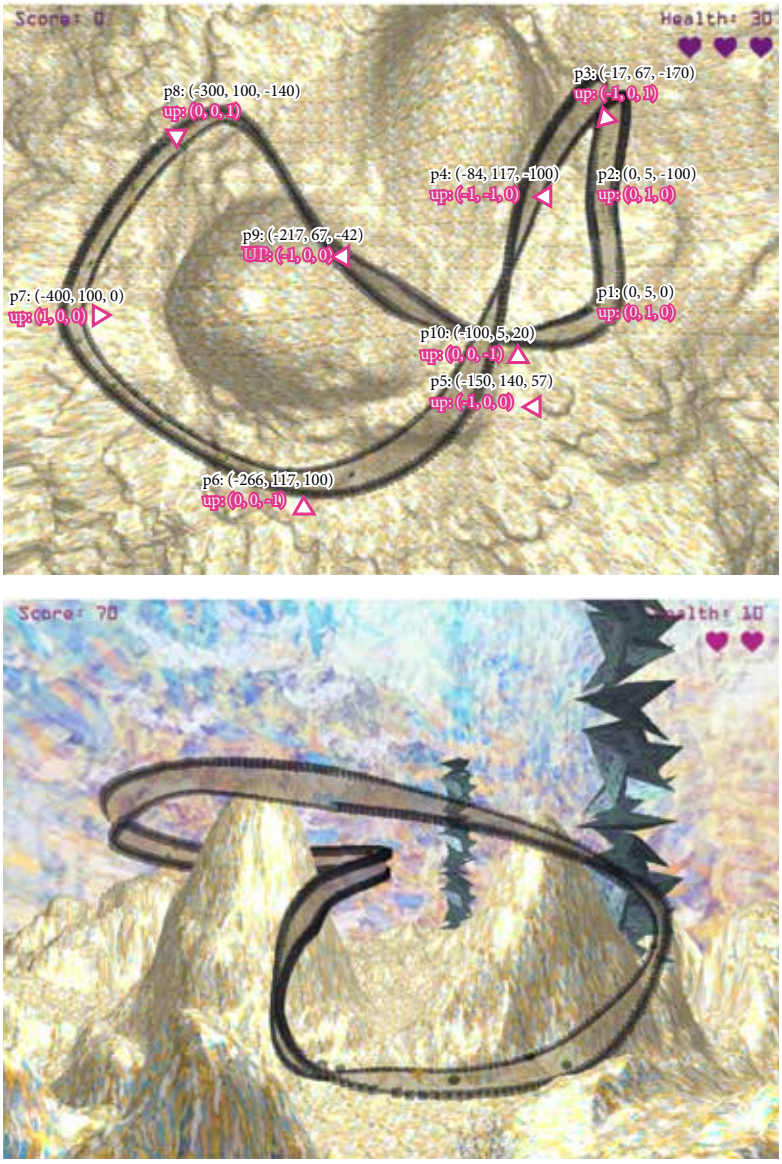


Figure 4: 3D spline for track, with offset-curve rendering turned on for visual clarity

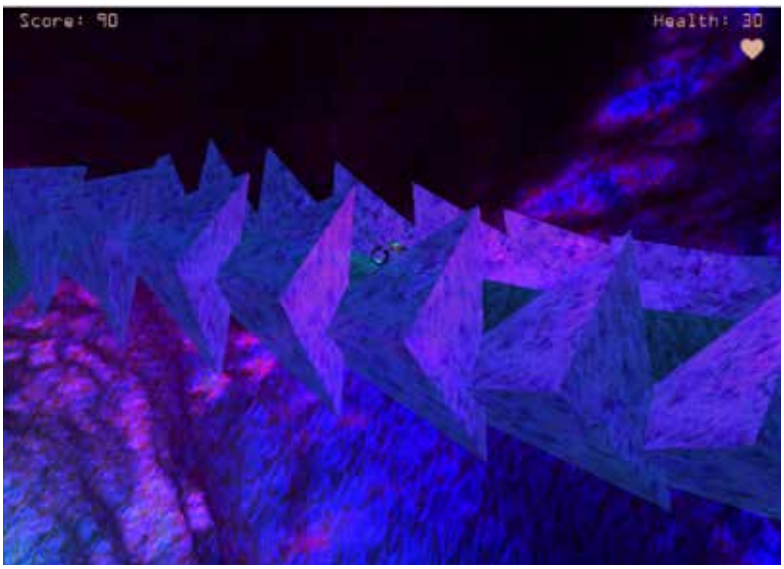
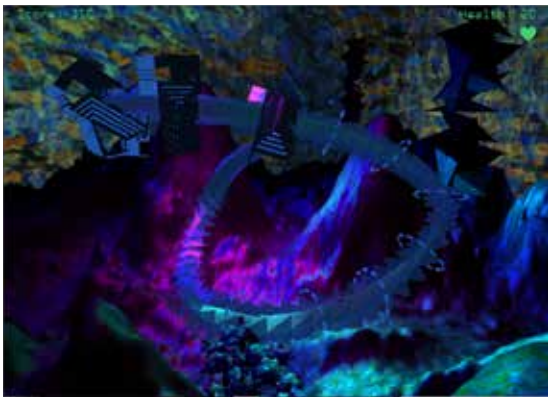


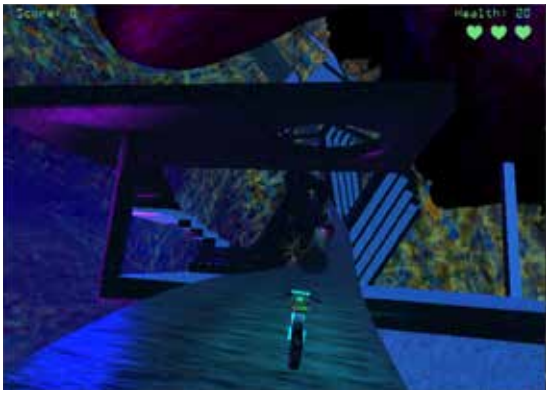
Figure 5: Side-view of tetrahedrons that demarcate the edge of the route.

II: Route and Camera

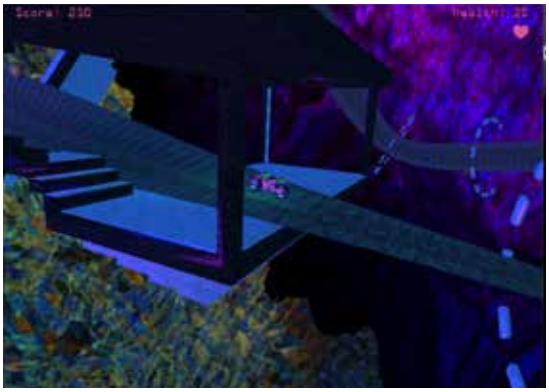
Figure 6: Overview of Camera Controls



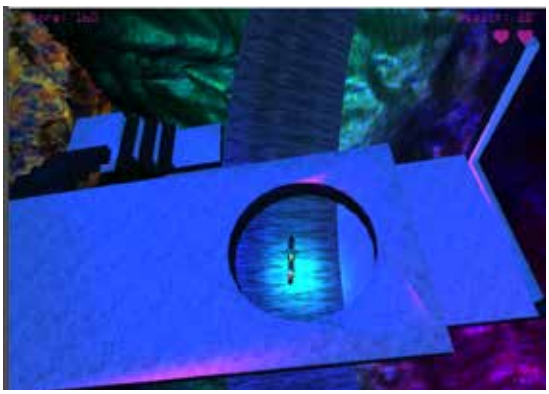
Key: 'F' Toggle Free View



Key: VK_F1 3rd Person View



Key: VK_F2 Side View



Key: VK_F3 Top View

The project enables four different types of camera views, as seen in *Figure 6*: free view, 3rd person, side view, and top-down view. The free view toggles independent camera control, where the camera is free to move anywhere in the scene. Pressing 'F' again will disable free view, attaching the camera onto the player - namely, the motorcycle.

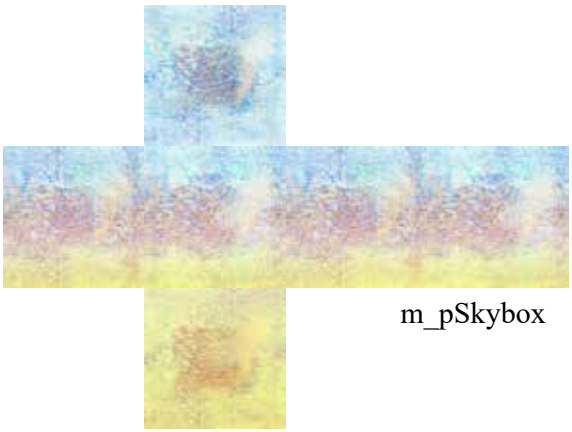
The camera viewing geometry is determined by the TNB frame, which has already been calculated in the CatmullRom class, and thus accessed in Game.cpp through getter methods. A helper function, Game::CameraControl(...), helps toggle between the different camera options: freeview, 3rd person view, side view, and top-down view.

Game.cpp

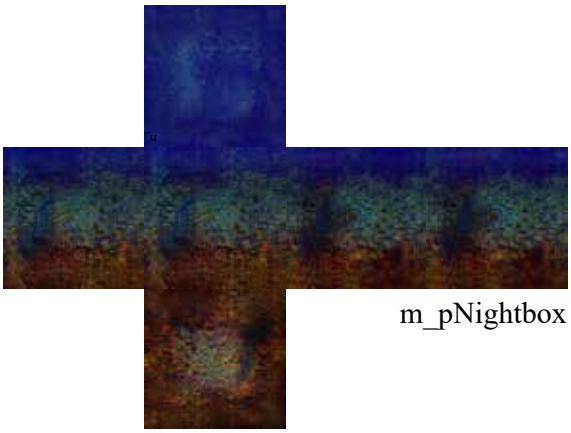
```
void Game::GameStart() //function in Game::Update()
{
    //Set Catmull Spline
    glm::vec3 p, up, pNext, pNextNext, upNext, upNextNext,
        playerUp, playerP;
    m_pCatmullRom->Sample(m_currentDistance, p, up);
    m_pCatmullRom->Sample(m_currentDistance + 25.0f, pNext,
        upNext);                //calculates T
    m_pCatmullRom->Sample(m_currentDistance + 26.0f, playerP,
        playerUp);              //calculates player position
    m_pCatmullRom->Sample(m_currentDistance + 50.0f, pNextNext,
        upNextNext);    //causes delay in camera rotation

    //Set 3rd person Camera (default)
    glm::vec3 T = glm::normalize(pNext - p);
    glm::vec3 P = p + up * 10.f;
    glm::vec3 viewpt = P + 10.0f * T;

    // Set Player
    glm::vec3 PlayerT = glm::normalize(playerP - pNext);
    glm::vec3 PlayerN = glm::normalize(cross(PlayerT, playerUp));
    m_pPlayer->Set(pNext, PlayerT, upNext);

   //Set Camera Options
   if (m_freeview == false) CameraControl(P, playerP, viewpt,
        PlayerN, playerUp, upNextNext);
   else { m_pCamera->Update(m_dt); }
}
```

```
void Game::CameraControl(glm::vec3& pos, glm::vec3& player,
glm::vec3& viewpt, glm::vec3& strafe, glm::vec3& up, glm::vec3&
upnext) {
    m_pPlayer->Update(m_dt); //player keys control
    switch (m_cameraControl) {
        case 1: { //third person camera
            m_pCamera->Set(pos, viewpt, upnext);
            break; }
        case 2: { //side view camera
        glm::vec3 camera_side_pos = pos - strafe * 50.f + up * 10.f;
            m_pCamera->Set(camera_side_pos, player, up);
            break; }
        case 3: { //top view camera
            glm::vec3 camera_top_pos = pos + up * 50.f;
            m_pCamera->Set(camera_top_pos, player, up);
            break; }
        default:
            m_pCamera->Set(pos, viewpt, upnext);
            break;
        }
}
```

Figure 7: Skyboxes for 'day mode' and 'dark mode'



m_pSkybox



m_pNightbox

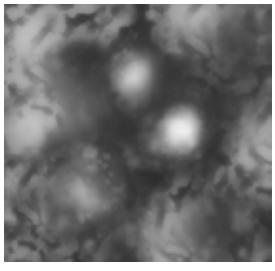
All requirements have been met in this section.

There are two lighting modes in the game scene: day mode and dark mode. The toggle between them is controlled by a boolean switch, `m_lightswitch`, and is accessed upon pressing 'tab'. In light mode, the game scene is lit by `light1`, the sunlight.
All spotlights are turned off. In dark mode, the sunlight is turned off and all spotlights are turned on. Material ambient light settings are set to 0 or otherwise small values in dark mode. Additionally, two different skyboxes are set for the different modes, as seen in *Figure 7*. They both derive from a modified class of Skybox.

The terrain is generated by using a heightmap. The heightmap is stored as a 2D image, where the colour of the pixel is interpreted by a height value y = f(x,z), and thus represents a displacement of a point on the plane - lighter pixels representing the higher zones, darker pixels representing the lower zones. A vertex mesh is created by feeding the resulting x,y,z information into the vertex list and then constructing triangles from the grid of points. The mesh is then texture-mapped, and the combination results as seen in *Figure 8* below.

texture

heightmap



resulting mountainous terrain

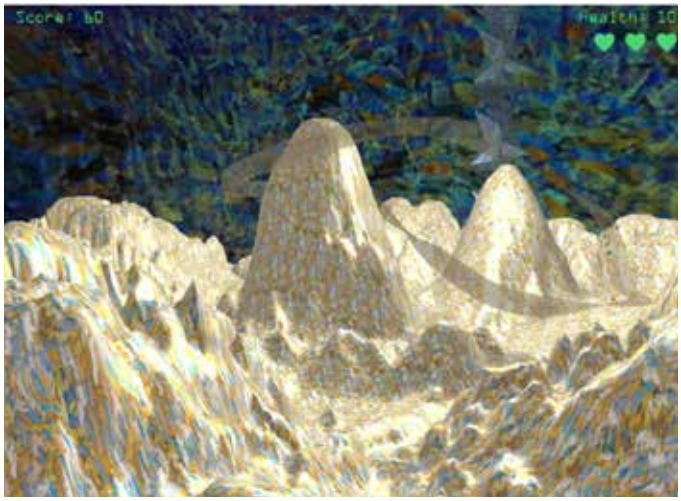
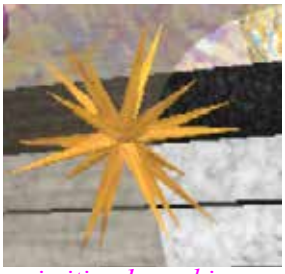


Figure 8: Generating a heightmap

Figure 9: Basic objects and Primitives.
To been seen in conjunction to Figure 1.

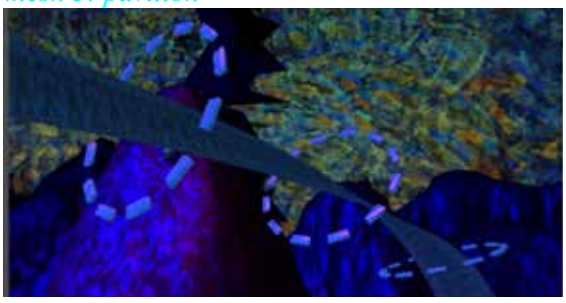

primitive 1: urchin


mesh 1: motorcycle


mesh 2: oak tree


primitive 2: tetrahedron


mesh 3: pavilion


mesh 4: rings

Two different basic objects are added from primitives: tetrahedron and urchin. They can be found under the 'basic shapes' folder under their respective classes, tetrahedron.h/.cpp and urchin.h/.cpp. Four different texture-mapped meshes are added into the scene: motorcycle, oak, pavilion, and ring. They use the OpenAssetImporter and are instantiated as ImportedAssetImportMeshes into the scene. Note that some .obj files were edited so that different parts of the mesh were texture-mapped using different textures, such as for the motorcycle and the oak tree. The oak tree rendering, which uses instanced rendering, will be explained further onwards, under Part III for advanced rendering techniques.
The tetrahderon, and is rendered using triangle primitive type GL_TRIANGLES. Appropriate texture coordiates and normals are fed into the VBO. These are rendered using the pMainShader, and is lit using BlinnPhong lighting. The models are transformed to their intended positions in the scene using matrix transformations. The objects and meshes listed in *Figure 9* (left) approximately correlate to the positions indicated in the overarching sketch depicted in *Figure 1*.

urchin.cpp (abbreviated snippets)

```
void Urchin::Create(string direct, string file, float spikes) {
    m_texture.Load(directory + filename);
    //set up VAO and bind VAO using helper classes, VBO/VBOI
    std::vector<glm::vec3>positions; //compute vertex pos
    std::vector<glm::vec2>texcoord; //compute texture coord
    std::vector<glm::vec3>normals; //compute face normals
    CVertex v0,v1,v2...  //compute vertices for VBO
    m_vbo.AddVertexData(&v0,sizeof_vertex);
    std::vector<GLunit>indices; //compute indices
    m_vbo.AddIndexData(&indices[0], sizeof(unsigned int);
    GLsizei stride;
    glEnableVertexAttribArray(0);
    glEnableVertexAttribPointer(...stride...);
}
```

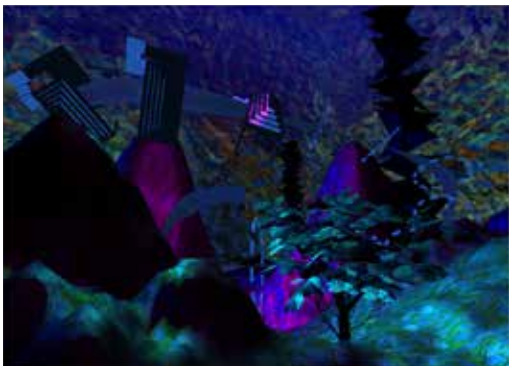
The abstract tower composed of tetrahedrons, are rendered into the scene using a for loop function that moves the tetrahedron position and rotation upon every iteration. Others, like ring mesh, use more complex quaternion transformations as they need to be rotated in three dimensions so that they align perpendicular to the track in the game scene. The urchin primitive - which is used as a 'bomb' in the game - is repeatedly rendered along random positions on the 3D spline. Additionally, they appear to rotate as the their rotational angle is constantly being updated by the amound of time elasped in the game.

Game.cpp (matrix transformations)

```
// Render the tetrahedron towers
   for (int i = 0; i < 20; i++) {
   modelViewMatrixStack.Push();
   modelViewMatrixStack.Translate(
        glm::vec3(x, y + i * 20.f, z));
    modelViewMatrixStack.RotateRadians(
        glm::vec3(0.5f, 1.f, 0.5f), (float)
        M_PI/(3) * i);
    modelViewMatrixStack.Scale(10.f);
    pMainProgram->SetUniform(
        matrices.modelViewMatrix",
        modelViewMatrixStack.Top());
    pMainProgram->SetUniform(
        "matrices.normalMatrix",
        m_pCamera->ComputeNormalMatrix(
        modelViewMatrixStack.Top()));
    m_pTetrahedron->Render();
    modelViewMatrixStack.Pop();
}
```

Figure 1: Different light modes, toggle 'TAB' to switch modes

Day Mode



Dark Mode

There are four spotlights in the scene that switch on in Dark Mode. Two spotlights of different colours - cyan and magenta - are static and are used to light up the world, and two spotlights dynamically follow the player to light up the motorcycle and the path ahead. Their locations and colour are illustrated in *Figure 11*. Note that in dark mode, ambient light is turned off and the world is lit up entirely through diffuse or specular light. The mainShader is responsible for most of the lighting, and uses the PhongModel lighting and BlinnPhong spotlight model respectively.

mainShader.frag

```
void main() {

    //Render world sunlight using the PhongModel implementation
    vec3 vColour = PhongModel(eyePosition, normalize(eyeNorm));

    //Render the four spotlights using BlinnPhongSpotlightModel
    for (int i = 0; i < 4; i++) {
        vColour += BlinnPhongSpotlightModel(
                spotlight[i], eyePosition, normalize(eyeNorm));
    }
    if (bUseTexture)
    //Combine object colour and texture
        vOutputColour = vTexColour*vec4(vColour, 1.0f);
}
```

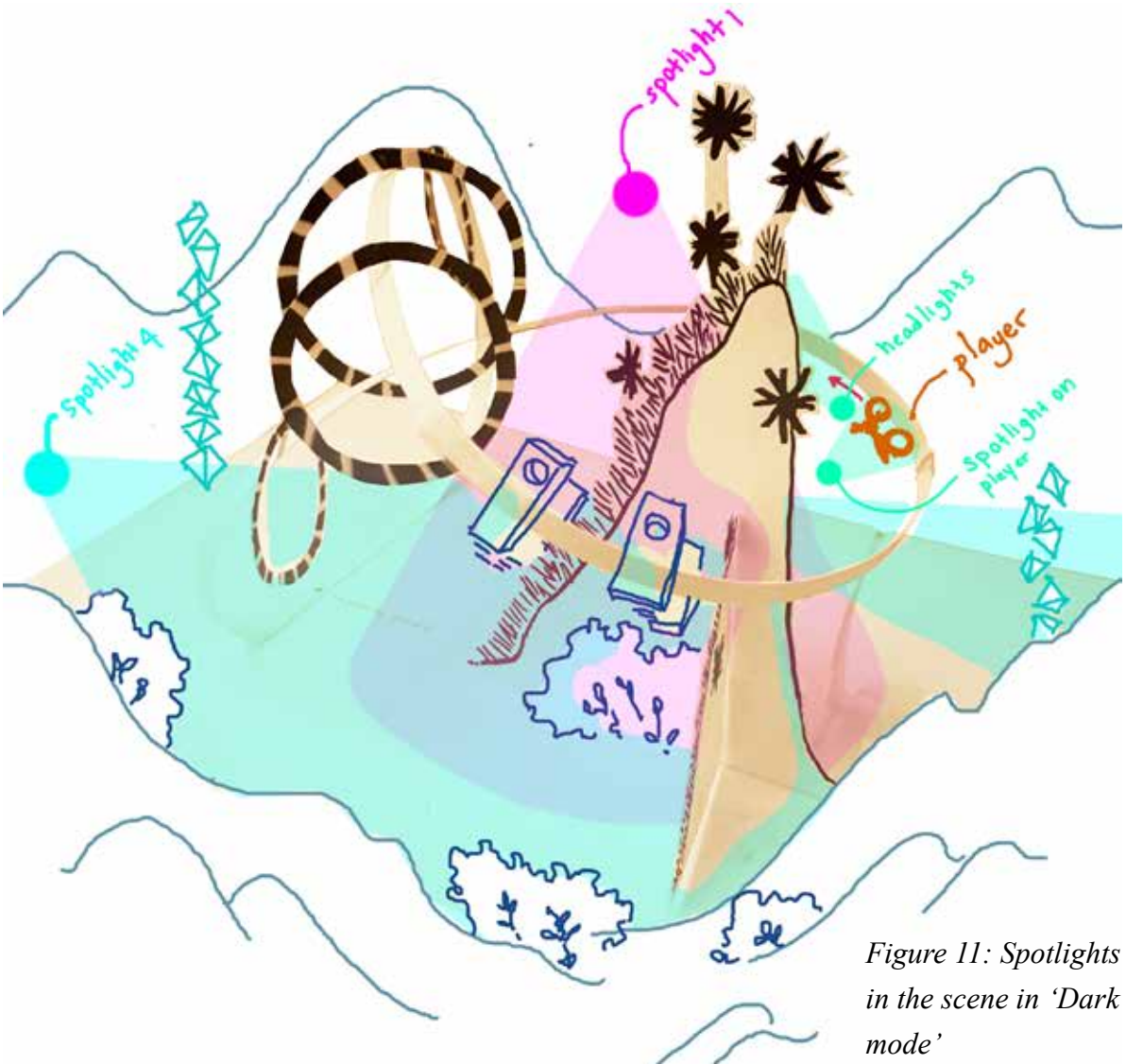
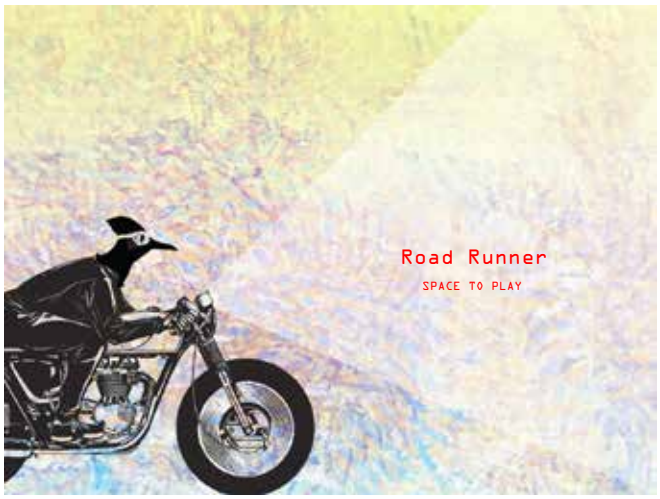


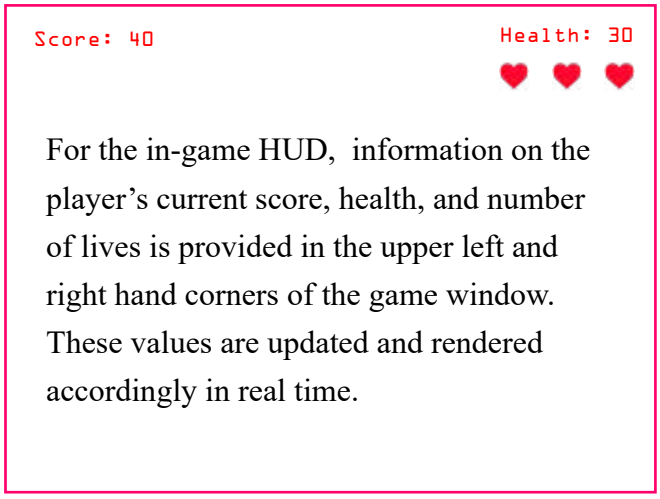
Figure 11: Spotlights in the scene in 'Dark mode'

Figure 12: HUD



Intro screen

Game over screen



For the in-game HUD, information on the player's current score, health, and number of lives is provided in the upper left and right hand corners of the game window. These values are updated and rendered accordingly in real time.

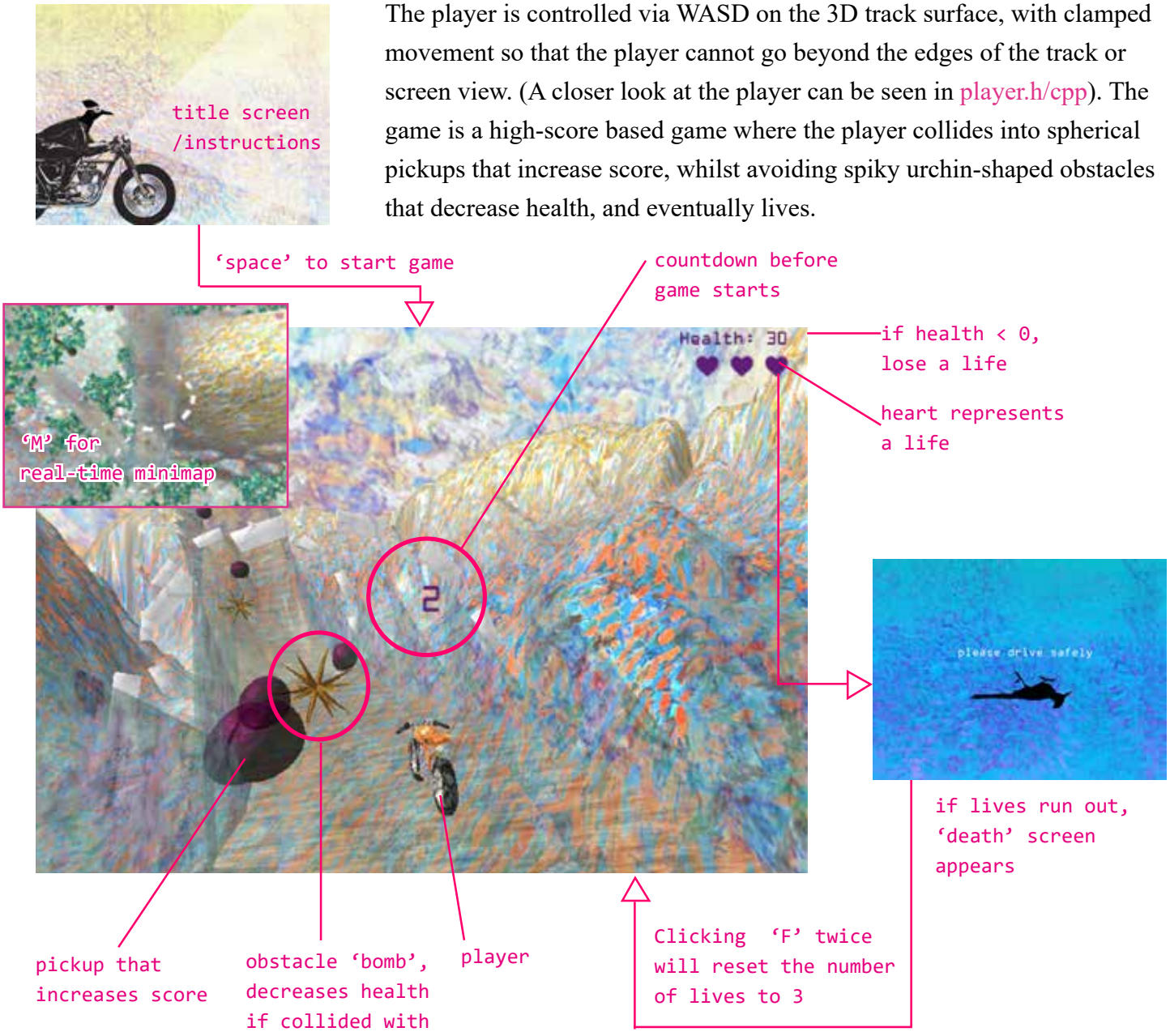
The HUD uses a modified implementation of the textShader and is rendered using the orthographic camera projection. Regarding the font rendering, the font is changed from Arial to OCR A Regular, and a time-based rainbow colour change is implemented - meaning that the text colour shifts from pink to green to blue and etc.

The plane class has been modified to enable transparent shading. This is accomplished by enabling GL_BLEND and setting the alpha blending settings to GL_ONE_MINUS_SRC_ALPHA. This enabled icons, such as the ❤ to be rendered without creating ■. In the textShader, this is enabled by homogenizing vTexColour.r across both RGB and Alpha, thus enabling transparency. For screens outside of the gameplay, ie the title screen or the game over screen, a texture was bound to a plane primitive, positioned and scaled to the four corners of the game window, and rendered using the orthographic camera.

```
//textShader.frag, used for objects rendered in
//orthographic perspective
void main() {
     //changes colour depending on time 't'
     vec4 rgbColour = vec4(0.5 + 0.4*sin(t), 0.5
         + 0.4* cos(t), 0.5 + 0.1*sin(t), 1);
     //samples the texture if any
     vec4 vTexColour = texture(sampler0, vTexCoord);

     if (bText)
     { //if rending rainbow mode text with alpha
       //blending background
          vOutputColour =
               vec4(vTexColour.r) * rgbColour;
     } else if (bRGB)
     { //if rendering rainbow mode image
          vOutputColour = vTexColour * rgbColour;
     } else //if rendering image
          vOutputColour = vTexColour;
}
```

The HUD and orthographic rendering code can be found in Game::DisplayHUD(...). Here, the shader program is switched to the fontProgram with the textShader.frag/vert information. GL_DEPTH_TEST is disabled as it is not needed for 2D rendering.

Figure 13: Gameflow

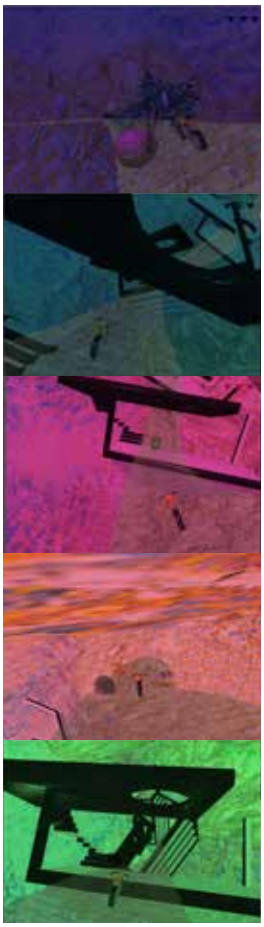


The player is controlled via WASD on the 3D track surface, with clamped movement so that the player cannot go beyond the edges of the track or screen view. (A closer look at the player can be seen in player.h/cpp). The game is a high-score based game where the player collides into spherical pickups that increase score, whilst avoiding spiky urchin-shaped obstacles that decrease health, and eventually lives.

A distance function - glm::distance - is implemented as the main collision function, causing the player to trigger pickups and bombs within close proximity. Separate classes for pickups (Pickup.h/.cpp) and bombs (Bomb.h/.cpp) are integrated into the project, with basic methods to initialize, render, and update these objects. In the game scene they are initialized as a vector of pickups (vector<CPickup*> *m_pPickups) and bombs (vector<CBomb*>m_pBombs) and instantiated along random positions along the track.

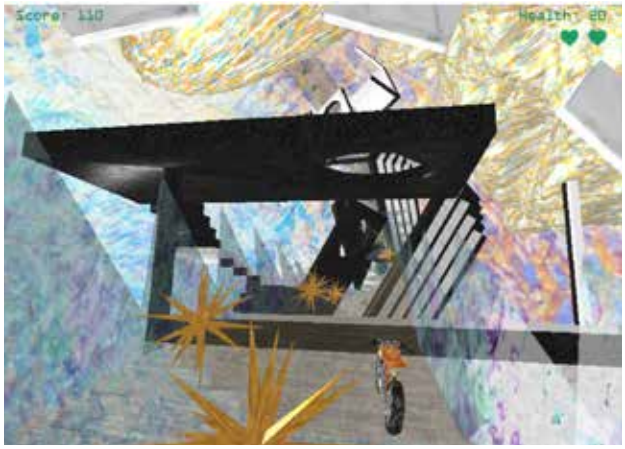
```
// Initialise Vector of Pickups
for (int i = 0; i < m_pickup_num; i++) {
     //generate random position along track
     int random = rand() % 400;   //random point of the 400 sampled points along track
     glm::vec3 pickup_pos =
          m_pCatmullRom->GetTrackPoints()[random] //random position along track length
            + (m_pCatmullRom->GetOffsetPoints()[random])*(rand()%20); //random position along track width
     //push back Pickup objects into vector of Pickups
     m_pPickups->push_back(new CPickup(m_pSphere, pickup_pos));
}
```

Figure: rainbow filter



Using the Frame Buffer Object (FBO), five different colour filters may be applied to the scene. Filters are accessed via KEYS 1, 2, 3, 4, and 5. The original scene is accessed via KEY 0. Note that the textShader has been modified to apply the colour filter.

Figure: Original scene (no filters).



textShader.frag

```
uniform int switchColour;

 if (switchColour == 0) //no filters: output colour is the texture colour
     vOutputColour = vTexColour;
 else if (switchColour == 1) //black and white: homogenize r value across rgb range
     vOutputColour = vec4(vec3(vTexColour.r), 1.0);
 else if (switchColour == 2) //colour shift: switch around rgb values as gbr
     vOutputColour = vec4(vTexColour.g, vTexColour.b, vTexColour.r, 1);
 else if (switchColour == 3) //vibrant: square values for rgb
     vOutputColour = vec4(vTexColour.r*vTexColour.r, vTexColour.g * vTexColour.g,
          vTexColour.b *vTexColour.b, 1);
 else if (switchColour == 4) //rainbow: run rbg values through time-base sine wave
     vec4 rgbColour = vec4(0.5 + 0.4*sin(t), 0.5 + 0.4* cos(t), 0.5 + 0.1*sin(t), 1);
     vOutputColour = vTexColour * rgbColour;
 else //inverted colours: invert rgb values by (1 - value).
     vOutputColour = vec4(1 - vTexColour.r, 1 - vTexColour.g, 1 - vTexColour.b, 1);
```

After the FBO is bound to the game scene, **`m_pFBO->Bind()`**, the scene is rendered as a texture using **`glBindFramebuffer(GL_FRAMEBUFFER, 0)`**. This texture is then bound to a plane primitive, **`m_pFilter`**, and rendered onto the screen using an orthographic camera. Note that the setup for screenspace rendering can be found under **`Game::DisplayHUD()`**.

```
m_pFBO->BindTexture(0);
m_pFilter->Render(false);
```

A helper function **`Game::ColourControl(...)`** has been set up on Game.cpp to link the desired shader option into the game scene.ColourControl( ) uses an int variable **`m_switchColour`**, controlled by key inputs 0-5, to determine the uniform int **`switchColour`** and thus select how the texture on m_pFilter is rendered.

```
void Game::ColourControl(CShaderProgram* fontProgram)
{
        fontProgram->SetUniform("switchColour", m_switchColour);
}
```

Figure: Light/Dark colour filter options 1-3 & 5

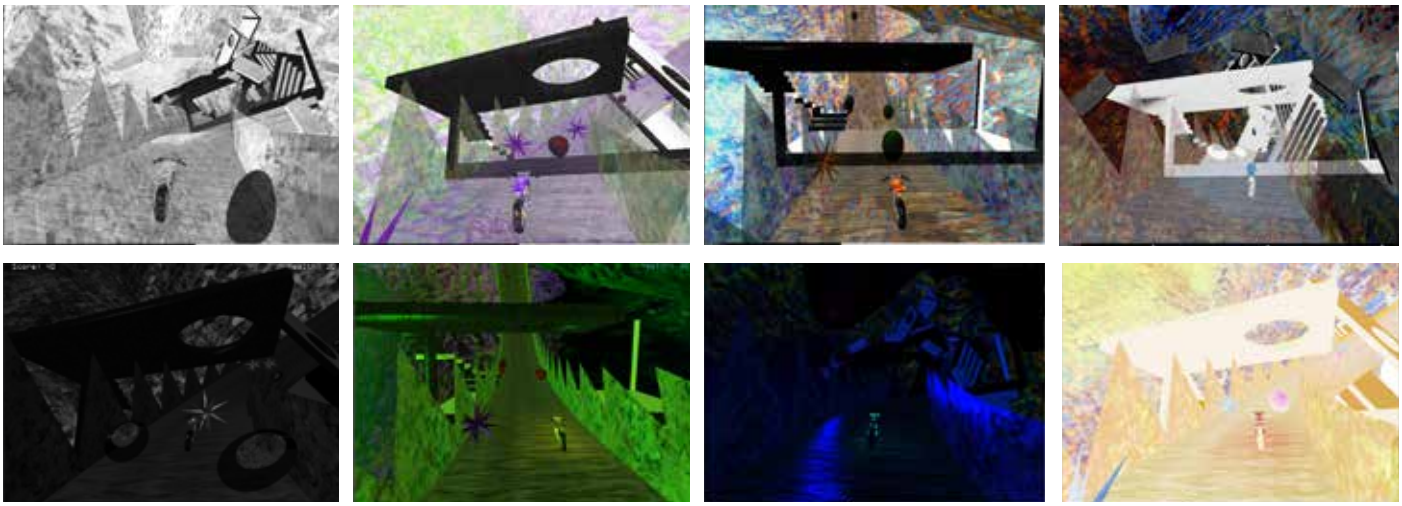
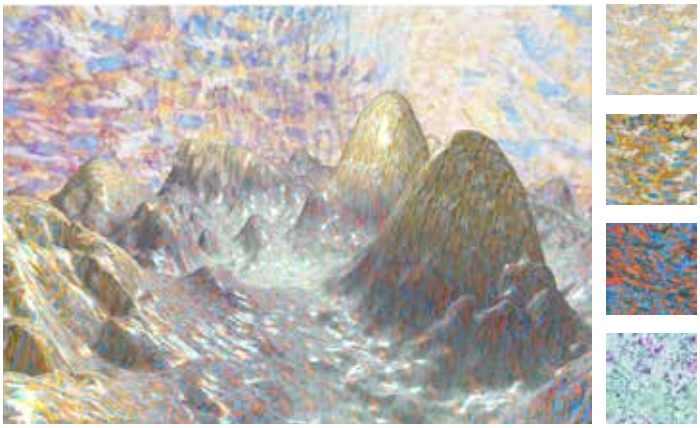
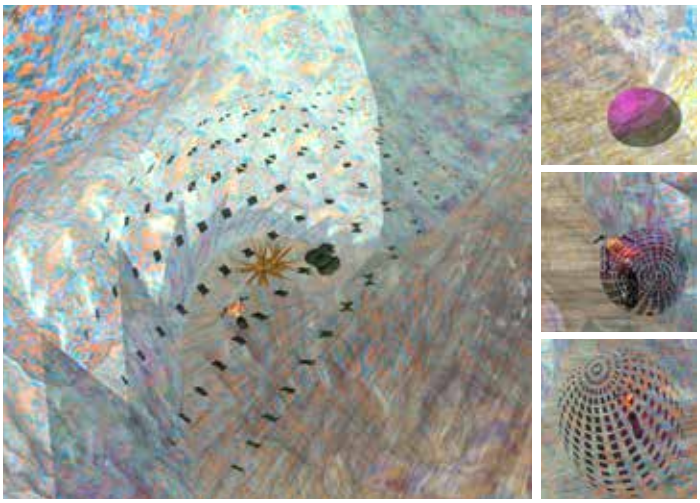


Figure: multi-texturing



Multi-texturing is used for the heightmap terrain, which is sampled across four different textures according to world height. This is triggered from the mainShader if uniform bool **`renderTerrain`** is true. The minimum **fMinHeight** and maximum **fMaxHeight** values determine which range of y-values corresponds to which texture. the `mix` function allows different textures to blend into each other to make the multi-texturing more natural. Then in HeightMapTerrain.cpp, the texures are loaded and bound using. The textures are then linked to the shader program by setting the uniform sampler2D sampler0, sampler1, and so on.

Figure: exploding pickups/bombs



In Bomb.cpp, **`explodeFactor`** is calculated as a time-based float that increases with elapsed time. For a more natural explosion effect, the factor should increase exponentially-- and is thus square.

A similar shader has been integrated to toonShader for pickups.

mainShader.frag

```
else if (renderTerrain) {
    vec4 vTexColour0 = texture(sampler0, vTexCoord);
    vec4 vTexColour1 = texture(sampler1, vTexCoord);
    vec4 vTexColour2 = texture(sampler2, vTexCoord);
    vec4 vTexColour3 = texture(sampler3, vTexCoord);

    float f = clamp(3*(worldPosition.y - fMinHeight) /
        (fMaxHeight - fMinHeight), 0, 3);

    vec4 vTexColour;
    if (f < 1)
        vTexColour = mix(vTexColour0, vTexColour1, f);
    else if (f < 2)
        vTexColour = mix(vTexColour1, vTexColour2,
           f - 1.0);
    else
        vTexColour = mix(vTexColour2, vTexColour3,
           f - 2.0);
        vOutputColour = vTexColour*vec4(vColour, 1.0f);
}
```

An **exploding geometry shader** is applied to render bombs, and is set to trigger upon contact (see *explodingShader.geom*). The exploding shader works by pushing all the triangles outward along their face normals. In this case, the VAO of the mesh object works with triangles **`layout(triangles).`** To find the normal, the cross product of **`e1`** and **`e2,`** two vectors in the face of a triangle, are calculated. A transformation along the normal by some scalar factor **`explodeFactor`** is then applied to the primitive --but only if uniform bool **`bExplodeObject`** is true.

explodingShader.geom

```
# version 400

in vec3 vColourPass[];out vec3 vColour;
in vec2 vTexCoordPass[]; out vec2 vTexCoord;

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

uniform bool bExplodeObject;
uniform float explodeFactor;

void main() {
    float localExplode = bExplodeObject?
        explodeFactor : 0.0;

    vec3 e1 = gl_in[1].gl_Position.xyz
              - gl_in[0].gl_Position.xyz;
    vec3 e2 = gl_in[2].gl_Position.xyz
              - gl_in[0].gl_Position.xyz;
    vec3 n = normalize(cross(e1, e2));

    for (int i = 0; i < 3; i++) {
        vec4 explodedPos = gl_in[i].gl_Position
            + vec4(localExplode * n, 0);
        gl_Position = matrices.projMatrix
            * matrices.modelViewMatrix * explodedPos;
        vColour = vColourPass[i];
        vTexCoord = vTexCoordPass[i];
        EmitVertex();
    }
    EndPrimitive();
}
```

The **toonshader** is a modified version of sphereShaderEd from Lab 4. The toonShader does three things: animates a 'bounce' in the vertex shader, 'explodes' its geometry in the geometry shader, and changes colour depending on proximity to the player. The shader changes the positions of the primitive's vertices as a sine wave, where the position along the wave is determined by elapsed time 't'. These positions are fed into the geometry shader, and exloded along its normals upon contact with the player. Lastly, in the fragment shader, the colour is quantised into layers by approximating to the nearest colour level, **`quantisedColour`**, thus producing a toon-like shading effect. The rgb values of the quantisedColour is shifted according to the proximity of the player to the pickup, which is scaled to a value between [0 1] outside the shader.

toonShader.vert, the "bounce" animation

```
void main() {
    vec3 p = inPosition;
    p.y += sin(p.z + t);
    p.z += sin(p.x + t);
    gl_Position =  vec4(p, 1.0);
    ...
}
```

toonShader.geom, the "explode"

```
//see previous page for explodeShader.geom
```

toonShader.frag, the "toon" rendering

```
uniform int levels;
in vec3 vColour;
uniform float changeColour;

void main() {
    quantisedColour = floor(vColour * levels) / levels;
    vOutputColour = vec4(vec3(
        quantisedColour.r + changeColour,
        quantisedColour.g,
        quantisedColour.b - changeColour/2),
        0.6f) ;
}
```

The treeShader allows for **instanced rendering**. Instanced rendering is useful for rendering many repeated objects with very little overhead.

Firstly, in the OpenAssetImportMesh. cpp, **`glDrawElements`** is changed to **`glDrawElementsInstanced`**. This is important as it allows the instances to be rendered in a single call using a GLSL shader. The render call is accessed via the function **`RenderInstances(int count),`** where 'count' allows the programmer to specify the number of rendered instances to be desired.

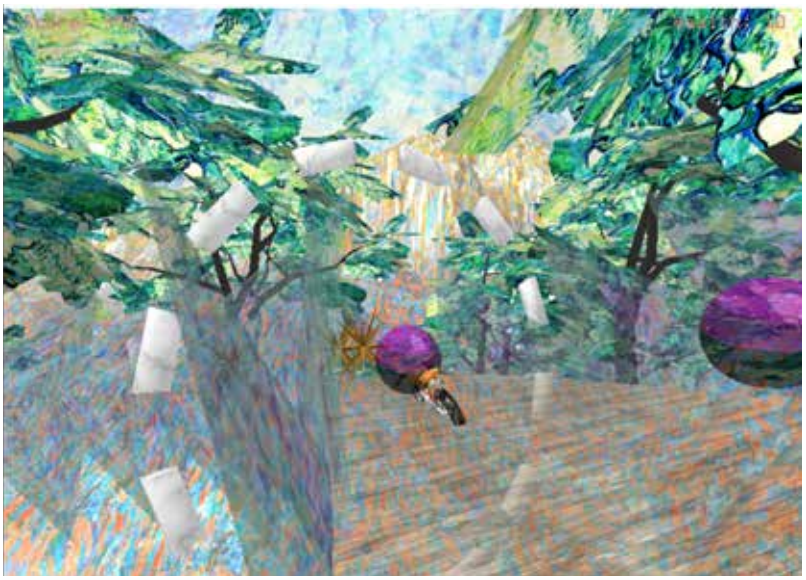
OpenAssetImportMesh.cpp

```
void COpenAssetImportMesh::
     RenderInstances(int count)
{
    glBindVertexArray(m_vao);
    for (unsigned int i = 0;
        i < m_Entries.size(); i++)
     {
        //do some preparations
        glDrawElementsInstanced(
            GL_TRIANGLES,
            m_Entries[i]. NumIndices,
            GL_UNSIGNED_INT,
            0, count);
    }
}
```

Game.cpp (RenderScene( ))

```
pTreeShader->UseProgram();
//do some preparations

modelViewMatrixStack.Push();
    m_pOakMesh->RenderInstances(30);
modelViewMatrixStack.Pop();
```

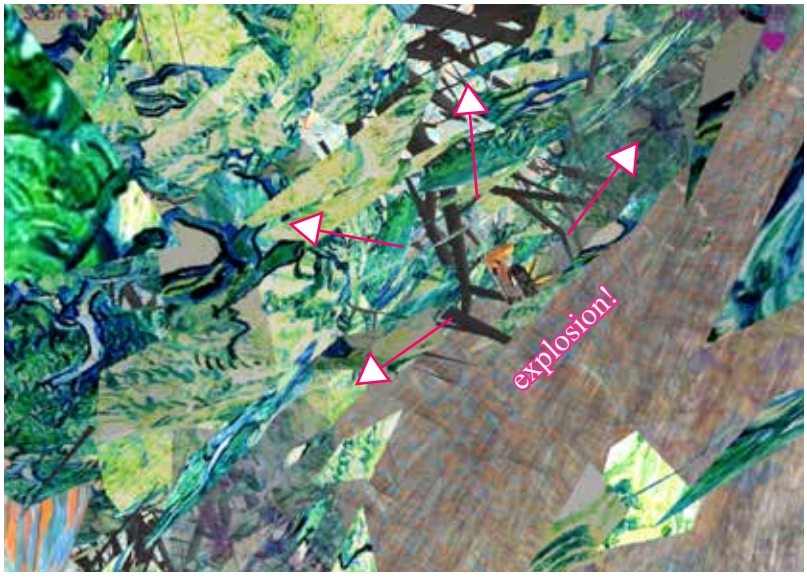
Figure: racing through a forest of instanced trees



treeshader.vert

```
void main()
{
        // Instanced rendering
        float x = gl_InstanceID % 6;
        float z = gl_InstanceID / 6 ;
        vec3 shift = 10 * vec3(x, 0, z);

        // Transform the vertex spatial position using
        gl_Position = vec4(inPosition + shift, 1.0f);

        // Get the vertex normal and vertex position in eye
        //coordinates
        eyeNormPass = normalize(matrices.normalMatrix * inNormal);
        eyePositionPass = matrices.modelViewMatrix *
            vec4(inPosition, 1.0f);

        // Pass through the texture coordinate
        vTexCoordPass = inCoord;
}
```

Figure: another explosion shader effect



GLSL has a built-in variable in the vertex shader called **`gl_InstanceID`**. gl_InstanceID relates to the instance being rendered for the rendering call, and allows each instance of the rendered object to be controllable. In the game, 30 instances, `m_pOakMesh->RenderInstances(30),` are made from each object. In the treeShader.vert, the gl_InstanceID is divided by six in both x and z directions, resulting in a grid of 6 x 5 = 30 instanced trees per tree object. The transformation between each instance is added as a vec3 shift in gl_Position.

Figure: Grid of Trees

Four trees are individually rendered on different parts of the game scene. A seen above, it creates a grid of uniformly spaced instances of each tree.

Note that near the end of the track, the player runs through a hanging branch that triggers an explosion through the tree's geometry shader, creating a flurry of leaves and bits of bark. This moment has been added to enhance the interactivity of the environment and demarcate the end of every lap. It has a similar implementation to *explodingShader.geom*, explained in the beginning of this subsection.

V: Conclusion

The goal of this project was to create a wacky and surreal racing game with fun and colourful graphics. Within the time provided, I believe that I have accomplished this goal. The final output of this project does include a wacky roller-coaster-like track that twists and turns in three dimensions, a composition of painterly and strange objects along the path, and various filters and lighting options of dizzying colours. However, there is also much room for improvement, particularly in regards to gameplay and advanced game rendering.

Regarding gameplay, I could include more levels of increasing difficulty. A single level in a racing game ends rather quickly, especially considering the speed of the player in the game. Small additions, like creating a wider range of pickups, would help diversity the gameplay choices. The current gamescene only offers pickups that increase score, but I could have also incorporated pickups that boost speed or health. I could also introduce a competitive element to the game by adding NPC racers along with the player, or alternatively a highscore board with other players who also play the game.

Regarding advanced game rendering, I would have liked to do more with the frame buffer object. Adding shadowmapping would help add more detail to the environment, and would look especially good in regards to the shadows created from the headlights of the player object in 'dark mode'. I would have also liked to include a SSAO shader for post-rendering effects to add to the realism of the scene. Furthermore, the forest could look a lot more lush and dense with a noise filter added to the instancing. Including more special effects, like particle effects or animations, would have greatly enhanced the dynamism of the game.

Nevertheless, I hope you enjoy racing in "Road Runner", a surreal world full of colour and strange surprises.

V: Assets List

All other assets created by Jihae Han and otherwise described to further detail in the OpenGLTemplate project.

"dirtpile" http://www.psionicgames.com/?page_id=26. Downloaded on 24 Jan 2013.
"DST-Canopy.mp3". Royalty Free License. http://www.nosoapradio.us/. Downloaded on Feb 4, 2020.
"Marble Texture". Freepik License. www.freepik.com. Downloaded on March 14, 2020.
"Motorcycle_logo". Freepik License. Background vector created by kojek90. www.freepik.com. https://www.freepik.com/free-photos-vectors/background. Downloaded on April 2, 2020.
"oak tree" Creative Commons. https://opengameart.org/content/oak-tree. Downloaded on March 19, 2020.
"Olive Grove" (1889) Museum photograph. Kröller- Müller Museum, Otterlo, The Netherlands. https://krollermuller.nl/en/vincent-van-gogh-olive-grove. Downloaded on Feb 20, 2020.
"motorcycle" Creative Commons. https://opengameart.org/content/harakiri-warbike-1000cc. Downloaded on March 21, 2020.
"Quaternions" http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/#how-do-i-create-a-quaternion-in-c-. Accessed Mar 10, 2020
"RoadRunner" Freepik License. https://www.flaticon.com/free-icon/bird-roadrunner-shape_47127. Downloaded on April 2, 2020.
"The Sower" (1888) Museum photograph. Kröller- Müller Museum, Otterlo, The Netherlands. https://www.nytimes.com. Downloaded on Feb 20, 2020.
m/2015/06/12/arts/design/review-van-gogh-and-nature-exploring-the-outside-world-in-high-relief.html