INF443 PROJECT

Treasure hunting in Ancient Egypt

June 2024

Gabriel Pereira de Carvalho
Otávio Ribas

CONTENTS

1
EXECUTIVE SUMMARY

Project: Treasure hunting in Ancient Egypt

The goal of our project is to recreate an archeological site in Ancient Egypt, roughly inspired by the Giza pyramid complex, trying to capture the distinct feel of this time period. To provide an incentive for exploration and experimentation with the controls, we scattered treasures across the map. In this way, the scene we designed can be experienced as a minigame: find all the hidden treasures in the map. The GUI includes a counter in order to track the number of treasures still missing.

In order to easily access the project's source code, please access the Github repository: `https://github.com/ArkhamKnightGPC/OpenGL_AncientEgyptProject`.

2
IMPLEMENTED FEATURES

2.1 Background music using SDL C++ library

We used the SDL library for playing a *.wav* file as background music. It is necessary to install this library in order to run the project locally. In Linux, this can be done with

```
    sudo apt-get install libsdl2-dev
```

The library provides a playback queue where we can push audio files to play simultaneous to the animation loop.

```
SDL_Init(SDL_INIT_AUDIO); // Initialize SDL
SDL_LoadWAV("../sounds/uncharted.wav", &wavSpec, &wavBuffer, &wavLength); // load .wav file
deviceId = SDL_OpenAudioDevice(nullptr, 0, &wavSpec, nullptr, 0); // open audio device
SDL_QueueAudio(deviceId, wavBuffer, wavLength); // queue file for playback
SDL_PauseAudioDevice(deviceId, 0); // play audio
```

In the project, once playback is finished and the queue is empty, we just replay our background music file.

```
// Wait for the sound to finish playing
if (SDL_GetQueuedAudioSize(deviceId) == 0) {
    // and play it again!!!!
    SDL_QueueAudio(deviceId, wavBuffer, wavLength);
    SDL_PauseAudioDevice(deviceId, 0);
}
```

2.2 Emulating sand surface with Perlin Noise

The ground surface in our project is modelled as a rectangular grid mesh. In order to add realism and emulate the accumulation of sand over this surface, we used perlin to add a randomised noise to the z coordinate of our mesh.

Perlin noise, allows smooth interpolation between noise values which gives the surface a very natural look. We stored the resulting heights in a C++ `std::map`. This map allows us to draw subsequent models with the correct z offset over the terrain.

```
void scene_structure::deform_terrain(mesh& m)
{
  for (int k = 0; k < m.position.size(); ++k){
    vec3& p = m.position[k];
    float z = noise_perlin({p.x, p.y});
    p = { p.x, p.y, z };
    height_map[std::pair<int, int>(p.x, p.y)] = z;
  }
  m.normal_update();
}
```

2.3 Camera controller and user interaction

In order to respond to keyboard controls and provide a user interactive experience, we developed a custom camera controller where the camera's displacements are controlled by the keyboard.

```
    //WASDQE used for camera rotations
    if (inputs->keyboard.is_pressed(GLFW_KEY_W))
      camera_model.manipulator_rotate_roll_pitch_yaw(0, pitch * magnitude,
 0);
    if (inputs->keyboard.is_pressed(GLFW_KEY_S))
      camera_model.manipulator_rotate_roll_pitch_yaw(0, -pitch * magnitude,
 0);

    if (inputs->keyboard.is_pressed(GLFW_KEY_A))
            camera_model.manipulator_rotate_roll_pitch_yaw(0, 0.0f, roll *
 magnitude);
    if (inputs->keyboard.is_pressed(GLFW_KEY_D))
      camera_model.manipulator_rotate_roll_pitch_yaw(0, 0.0, -roll *
 magnitude);

        if (inputs->keyboard.is_pressed(GLFW_KEY_Q))
            camera_model.manipulator_rotate_roll_pitch_yaw(roll * magnitude,
  0.0f, 0.0f);
        if (inputs->keyboard.is_pressed(GLFW_KEY_E))
            camera_model.manipulator_rotate_roll_pitch_yaw(-roll * magnitude
 , 0.0f, 0.0f);
```

```

    //IJKL used for camera translations
    if (inputs->keyboard.is_pressed(GLFW_KEY_I))
      camera_model.manipulator_translate_front(100*magnitude); // forward
    if (inputs->keyboard.is_pressed(GLFW_KEY_K))
      camera_model.manipulator_translate_front(-100*magnitude); // backward

    if (inputs->keyboard.is_pressed(GLFW_KEY_L))
      camera_model.manipulator_translate_in_plane({-100*magnitude ,0 }); // right
    if (inputs->keyboard.is_pressed(GLFW_KEY_J))
      camera_model.manipulator_translate_in_plane({100*magnitude ,0 }); // left
```

The complete list of controls is:

- Translation controls
  - **I** to move forward
  - **K** to move backward
  - **J** to move left
  - **L** to move right
- Rotation controls
  - **W** to rotate upward
  - **S** to rotate downward
  - **A** to turn left
  - **D** to turn right
  - **E** to turn counterclockwise
  - **D** to turn clockwise
- **R/F** to increase/reduce speed
- **H** to pick a treasure

The bird model representing the character controlled by the user follows the camera's movement. We also apply the camera's rotations to the bird model to give the impression of a natural flight pattern.

```
void scene_structure::idle_frame()
{
  camera_control.idle_frame(environment.camera_view);
  rotation_transform aux = camera_control.camera_model.orientation();
  bird_mesh.rotateBirdMesh(aux.matrix());
}
```

2.4 Modelling

The models used in our project include:

- Pyramid model (easily represented with geometric primitives)
- Bird model (developed for TD)
- Palm Tree model (given in base project)
- Sphinx, Camels, Treasure models (downloaded from Sketchfab)

In order to make the trees fairly spaced in the scene, we sampled their positions using a uniform distribution and checked for overlaps in order to avoid collisions between the models.

```
for(int i=0; i<100; i++){
    // Generate random tree coordinates
    int x_rand, y_rand;
        generateRandomCoordinates(x_rand, y_rand);
    //lets make sure no two trees overlap
    bool chk = true;
    for(int j=0; j<x_rand_trees.size() && chk; j++){
      double dist = pow(x_rand_trees[i] - x_rand, 2) + pow(y_rand_trees[i] -
    y_rand, 2);
      if(dist < 2000){
        chk = false;
      }
    }
    if(chk){
      x_rand_trees.push_back(x_rand);
      y_rand_trees.push_back(y_rand);
    }else{
      i--;
    }
  }
```

For the remaining models, we carefully applied the desired transformations in order to draw them in the positions we imagined to get the desired scene.

2.5 Treasure hunting minigame

Each treasure positioned in the scene is an instance of the *Treasure class*. This class is composed of two different meshes:

```
mesh_drawable glowing_sphere;
mesh_drawable reward;
```

At the start, each treasure is rendered as a white sphere which must be identified and picked by the player. While the pick action is not performed within a sufficiently small radius of the treasure, we consider it not found and continue displaying the white sphere.

```
void Treasure::drawSphere(environment_structure &environment){
    if(found == false){//we show sphere while treasure is not found
        draw(glowing_sphere, environment);
    }
}

void Treasure::pickTreasure(vec3 camera_position){
    double dist = pow((position[0] - camera_position[0]), 2) + pow((position[1] - camera_position[1]), 2) + pow((position[2] - camera_position[2]), 2);
    if(dist < 1000){//player is close enough to pick this treasure!!
        found = true;
    }
}
```

Once the treasure is picked we display a 3D model of the treasure over the character's head for a certain interval of frames (controlled by a countdown timer). In order to show multiple angles of the object we apply a periodic rotation controlled by a timer.

```
void Treasure::drawReward(environment_structure &environment, vec3 camera_position, mat3 camera_orientation){
    if(found && countdown_display > 0){
        countdown_display -= 1;

        //we will show the reward in front of the camera
        vec3 relative_position = camera_orientation*vec3(0, model_zoffset, -10);
        vec3 camera_vec = camera_position+relative_position;
        reward.model.translation = {camera_vec[0], camera_vec[1], camera_vec[2]};

        //androtate it so player can have a good look
        timer.update();
        const float minAngle = -M_PI / 6;
        const float maxAngle = M_PI / 6;
        float scaled_time = 3*timer.t;
        float angle = minAngle + (maxAngle - minAngle) * 0.5f * (1 + sin(scaled_time));
        reward.model.rotation = rotation_axis_angle({0,0,1}, angle);

        draw(reward, environment);
    }
}
```

In the file *scene.cpp*, the treasures are stored in a `std::vector`. In order to add a new treasure to the scene, we provide paths to its *.obj* file, its *.png* texture, a scaling factor and its position in the scene.

```
treasures.push_back(Treasure(
    project::path + "assets/treasures/bracelet.obj",
    project::path + "assets/treasures/bracelet.png",
    1,
    vec3{0, 0, 1.6+height_map[std::pair<int, int>(0, 0)]}));
```

In order to draw or interact with each individual treasure, we must iterate through this vector.

```
//inside display_frame function
for(auto &treasure : treasures){
    treasure.drawSphere(environment);
    treasure.drawReward(environment, camera_position);
}
```

```
void scene_structure::keyboard_event()
{
  if (inputs.keyboard.is_pressed(GLFW_KEY_H)){//player tried to pick a
   treasure!!
    vec3 camera_position = camera_control.camera_model.position_camera;
    for(auto &treasure : treasures){
      treasure.pickTreasure(camera_position);
    }
  }
  camera_control.action_keyboard(environment.camera_view);
}
```

Similarly, the counter shown in the GUI is computed iterating through this treasure list.

```
int cnt = 0;
for(auto &treasure : treasures){
    if(treasure.get_found())cnt++;
}
ImGui::Text("Treasure collected: %d out of %d", cnt, (int)treasures.size());
```

2.6 Phong Shader and Fog effect

In order to emulate a sandstorm effect over our desert, we used the fog effect developed in class. Even though the scene itself has a limited size, the limited render distance used in the shader gives the impression of a much bigger scene and also makes the treasures hidden in the map harder to find.

```
// Compute the base color of the object based on: vertex color, uniform
   color, and texture
  vec3 color_object  = fragment.color * material.color * color_image_texture
   .rgb;

  // Compute the final shaded color using Phong model
  float Ka = material.phong.ambient;
  float Kd = material.phong.diffuse;
  float Ks = material.phong.specular;
  vec3 color_shading = (Ka + Kd * diffuse_component) * color_object + Ks *
   specular_component * vec3(1.0, 1.0, 1.0);

  //FOG EFFECT
    float dist_to_camera = length(camera_position - fragment.position);
    float max_dist = 400;
    float fog_coeff = min(dist_to_camera/max_dist, 1.0);

    vec3 current_color = (1 - fog_coeff)*color_shading + fog_coeff*fog_color
   ;

  // Output color, with the alpha component
  FragColor = vec4(current_color, material.alpha * color_image_texture.a);
```

3 CONCLUSION

Developing this project has allowed us to better explore the techniques studied in class and, above all, to exercise our creativity while figuring solutions for the troubles found during its execution. There are, hence, a myriad of ideas on how to further develop it. The sensation of being surrounded by sand could be further enhanced by the incorporation of particles around the fixed camera. Additionally, one could always make the scenario more lively by incorporating animated creatures into it - like camel models capable of moving, human characters and other animal. Finally, a collision detector would be necessary to perfect the fantasy of being a bird amongst ancient ruins.