Middle East Technical University Department of Computer Engineering

CENG 477

Introduction to Computer Graphics

Fall 2023-2024
Assignment 3 - OpenGL with Programmable Shaders
Bunny Run
(v1.0)

Due date: January 5, 2024, Friday, 23:59

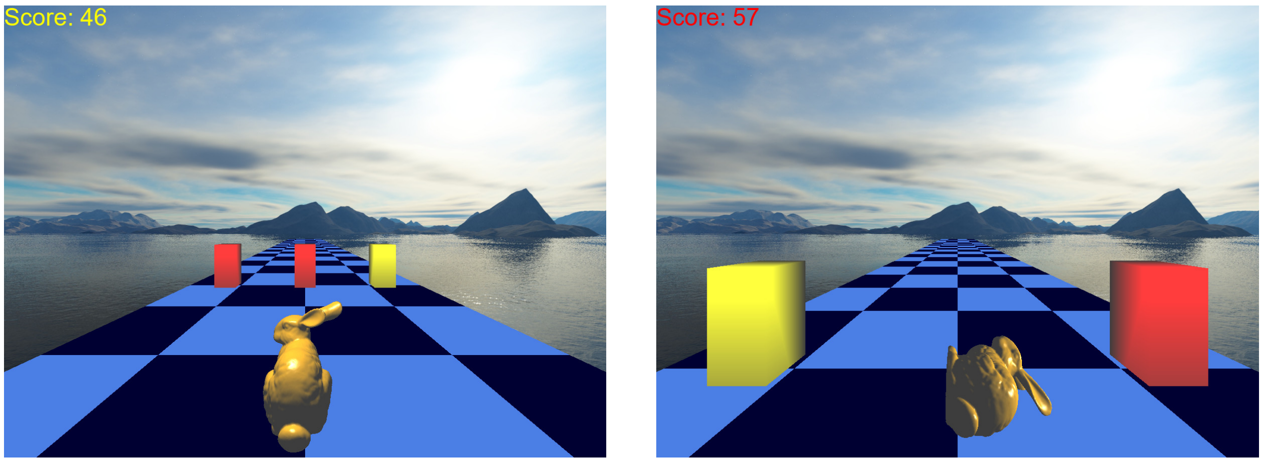

Figure 1: Sample visuals from Bunny Run.

1 Objectives

In this assignment, you will implement an OpenGL game called Bunny Run. In this game, the user controls a hopping bunny in the horizontal direction to capture the yellow checkpoints while avoiding the red checkpoints in an everlasting journey on a wide road to obtain a high score. The bunny continuously hops forward at an increasing speed, which makes it hard to keep the bunny alive as time passes in a round. Hitting one of the red checkpoints ends the round by making the bunny faint and lie down. Sample visuals that demonstrate this moment and also the game's scene composition are provided in Figure 1. More visuals can be found in the YouTube video we shared.

2 Specifications

Bunny:

- The bunny starts its journey at some forward speed, which increases slowly at each frame, making the bunny go forward faster.
- The bunny has a hopping animation that continues until the end of a round. Hopping is basically a linear go-up-and-down motion on the vertical axis. The quickness of the hopping animation increases as the bunny's forward speed increases at each frame.
- The player can move the bunny towards the left and right. The horizontal movement speed at which the bunny slides to the left and right increases as the bunny's forward speed increases at each frame so that the player can keep up with the speed of the incoming checkpoints during the fast moments of a round.
- The exact values of those parameters are up to your design.

Checkpoints:

- There are 3 checkpoints in total, which are positioned near each other on the road, and they approach the bunny all together. When their positions fall behind the positions of the bunny or the camera by some offset, they are re-positioned to the horizon and start coming towards the bunny again, simulating a flow of incoming checkpoints.
- Checkpoints have 2 types: *yellow* and *red* checkpoints. There is 1 yellow checkpoint and 2 red checkpoints. At each re-positioning of the checkpoints, the yellow checkpoint is randomized amongst the three.
- Hopping above or under the checkpoints is not possible. If a checkpoint is in the same lane as the bunny when they are very close to each other, that checkpoint is considered to be hit by the bunny. However, the bunny can be aligned by the user controls to move forward between the checkpoints without hitting any of them (skipping). This situation can be seen in the YouTube video.
- When a checkpoint is hit, it will no longer be drawn onto the screen until it gets re-positioned in the horizon. This situation can be seen in the YouTube video and in Figure 1.
- When the yellow checkpoint is hit, the bunny gets into a *happy state* in which it rotates around its vertical axis for one complete tour while still continuing to hop forward. The state ends after one complete tour ends. The rotation speed increases as the bunny's forward speed increases so that the happy state is completed before the next wave of checkpoints arrives at the bunny's position.
- When the bunny runs into a red checkpoint, it *faints* and lies down on the ground onto its right side, and the round ends. At that point, the user's inputs should not move the bunny anymore until a restart round command comes.
- A checkpoint is basically a cube object which gets scaled up in Y axis and translated into its world position.
- The exact values of those transformations and parameters are up to your design.

Camera:

- The camera follows the bunny's current Z position by some Y and Z offsets at each frame. It gets positioned higher in the Y axis and a tad behind in the Z axis when compared to the bunny's position. The exact values of those transformations are up to your design. It looks at the bunny's direction, which is the -Z direction in the shared video.
- The camera does not jump together with the bunny, and it does not move to the left or right when the bunny is instructed to do so by the user.
- Perspective projection will be used to render the scene composition. The video we shared uses the value 90 degrees as its FovY, the current window's width/height ratio as its aspect ratio, 0.1 as the near distance, and 200 as the far distance.

Ground:

- Ground is a quad mesh that is appropriately rotated to lie in the X-Z plane and scaled up in X and Z axes to have some width in the X axis and have an largely extended length in the Z axis. Moreover, to lie right under the feet of the bunny, it can be translated downwards a tad on the Y axis. The exact values of those transformations are up to your design.
- The quad mesh can be translated together with the position of the bunny mesh so that it feels infinitely enlarged. In reality, it will stick to the Z position of the bunny and move together with it.
- As the perspective projection will be used, the enlarged ground mesh will look like it extends to the horizon in the Z axis.
- The ground mesh must have a checkerboard color pattern on it, which is detailed in the Shaders section.

Score Text:

- At the top-left corner of the screen, the current score text must be visible in yellow color in the format: "Score: <INTEGER>". Even after resizing the window, the text should stay at the top-left corner.
- Current score is an integer value that starts at 0 and is increased at each frame by approximately the distance the bunny moved in Z axis so that as the bunny moves the score continuously increases.
- When the bunny hits the yellow checkpoint, the current score is increased by 1000.
- The scoring mechanism stops counting and shows the current score in red color after a red checkpoint is hit.
- The sample text rendering codebase that we shared can be leveraged in the homework.
- Any font that is installed in Inek machines can be used as long as it is readable on the screen.

Sky Texture (Bonus. 5 points):

- This is an optional step of the homework. Correctly drawing a textured sky using the given texture image file and the quad object will grant you a 5-point bonus.

- You can read the given texture image file from the disk using the stb_image.h library as follows:

  1. Download and include the file:

     ```
     #define STB_IMAGE_IMPLEMENTATION
     #include "stb_image.h"
     ```

  2. Load the image:

     ```
     int width, height, nrChannels;
     unsigned char *data = stbi_load(img_name.c_str(),
                           &width, &height, &nrChannels, 0);
     ```

- This image data is then set as a texture and sampled in the fragment shader of the sky quad.
- You can draw the sky quad first and also leverage the *glDepthMask* function to disable writing into the depth buffer while drawing the sky quad to make it stay as a background without occluding any of the meshes.

Shaders:

- The bunny's shaders must implement diffuse and specular color calculations, where the specular effect should be visible on top of the diffuse color on the surface of the bunny mesh in the scene. Those calculations should result in a yellowish bunny, as can be seen in Figure 1.
- The checkpoints' shaders must implement diffuse and specular coloring by having each checkpoint lit by a light whose position can be defined in the shader. The specular shading effect does not need to be perfectly visible. The critical bit is to have bright diffuse colors of yellow and red to make the checkpoints stand out in the scene with their colors. By using diffuse coloring, the realism of the checkpoint increases as it is lit well from its light position in its shader instead of returning a simple yellow color at each of its fragments. You can use the same shader files for all checkpoint types, but let the shader know the type of the checkpoint mesh it belongs to by using uniforms and make the shader result in the expected color for that checkpoint: Yellow or red.
- The ground's shaders must implement a checkerboard pattern coloring scheme. A checkerboard pattern can be created using the following pseudo-code:

  ```
  bool x = (int) ((pos.x + offset) * scale) % 2;
  bool y = (int) ((pos.y + offset) * scale) % 2;
  bool z = (int) ((pos.z + offset) * scale) % 2;

  bool xorXY = x != y;

  if (xorXY != z)
      return black;
  else
      return white;
  ```

Here, the *pos* variable is the 3D point at which we are calculating the texture value. *offset* controls the starting point of the pattern. By changing the offset, you can shift the pattern in the 3D space. *scale* controls the size of the checkerboard squares. *black* and *white* define the colors of the checkerboard squares – they do not have to be black and white: You can choose a color tuple that suits your taste.

- As the sky quad is left optional (bonus) in this homework, the background needs to be set to black color (0, 0, 0) via glClearColor function if you will not implement the sky texture.

Hints:

- We advise you to start early by inspecting the *sampleGL* codebase we shared with you on ODTUClass course page to see how it implements an OpenGL program. You can use it as the template code and build your homework code on top of it. You can also inspect the **basic_shader_glfw_model_text** codebase and leverage it in the text rendering part.
- You may want to have separate fragment and vertex shaders for each mesh. The default shaders of the sampleGL codebase can be leveraged as a starting point.
- A light's position in a shader can be multiplied with the *modelingMatrix* to make it look like the light is following the mesh when it moves in the 3D world. Lights in the bunny's shader and checkpoints' shaders, for example, can leverage this idea.
- As the bunny is continuously jumping, including the Y axis in the bunny's distance-to-the-checkpoints calculations may cause additional complexity. Therefore, checking the distance of the bunny to the checkpoints only on the X-Z plane at each frame can work well as a simple collision detection algorithm.
- As there might be multiple frames in which we are still colliding with the same checkpoint, you should handle the collisions by keeping some flags in your code to make the same collision occur only once. Else, hitting a checkpoint might give you +1000 score more than once in a moment in those sequential frames that come right after the first hit's frame.

User Controls:

- The program should have the following user controls (their effects can be seen in the demo video we shared, and pressing-and-holding a button should activate that button at each frame until it is released.):
  - **A and D buttons** should make the bunny move to its left and right, respectively. The bunny should not go out of the boundary of the road towards the left and right, that is, it should stay at a limit position from which it cannot go further left or right even though the appropriate direction button is still pressed.
  - **R button** should restart the round by re-initializing the values of your program, such as resetting the speed to its initial slow value, and re-positioning the checkpoints in the horizon. The yellow checkpoint's position gets randomized at each restart. Note that pressing and holding this button causes a restart at each frame. This effect can be seen in the video we shared.

3 Regulations

- **Programming Language:** C++
- **Late Submission:** You can submit your codes up to 3 days late. Each late day will be deduced from the total 7 credits for the semester. However, if you fail to submit even after 3 days, you will get 0 regardless of how many late credits you may have left. If you submit late and still get zero, you cannot claim back your late days. You must e-mail the assistant if you want your submission not to be evaluated (and therefore preserve your late day credits).
- **Cheating: We have zero tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations and will get 0 from the homework. You can discuss algorithmic choices, but sharing code between students is strictly forbidden. Please be aware that there are "very advanced tools" that detect if two codes are similar.
- **Additional Libraries:** GLM, GLEW, GLFW, stb_image.h, and FreeType libraries are the typical libraries you will need. You should not need to use any other library. If you still want to use some other library, please first ask about it in the ODTUClass forum of the homework.
- **Forum:** Any updates and discussions regarding the homework will be on ODTUClass. You should check it on a daily basis. You can ask your homework-related questions on the forum of the homework on ODTUClass.
- **Submission:**
  - Submissions will be done via ODTUClass. You can team-up with another student, or implement the homework on your own.
  - **Please be sure**, before submitting, that your code compiles and runs as you expect on Inek machines, as the frame rate of Ineks might not be as high as your own computer if you have implemented the homework on your own PC. This might cause the bunny move too slow on Ineks, requiring you to tune some parameters, such as the bunny's run speed, accordingly in your code.
  - You will provide a **Makefile** in your submission so that we can build your code on Ineks.
  - Create a **.zip** file that contains your source files, shader files, meshes, texture images and stb_image.h file (if sky texture was implemented), and a Makefile. It should not include any sub-directories.
  - If your directory structure is wrong or your Makefile does not work (or you do not have one) and therefore we will have to manually try to compile your code - there will be an automatic **penalty of 10 points** for each.
  - The name of the .zip file will be:
    * If a student works with a partner student:
      `<partner_1_student_id>_<partner_2_student_id>_opengl.zip`
    * If a student works alone:
      `<student_id>_opengl.zip`

        * For example:

          ```
          e1234567_e2345678_opengl.zip
          e1234567_opengl.zip
          ```

    - Make sure that when the commands below are executed, your executable file **"main"** is ready to use:

      ```
      >  unzip e1234567_opengl.zip -d e1234567_opengl
      >  cd e1234567_opengl
      >  make
      > ./main
      ```

    - **Therefore, you HAVE TO provide a Makefile in your submission.**

- **Grading:** Your codes will be evaluated on Inek machines. We will not use automated grading but will evaluate your outputs visually by playing your game. There is no frames-per-second (FPS) constraint in the homework but it should play smoothly in the Inek machines. We expect that in your implementation, the bunny moves at a speed that we show in the video (not too slow, not too fast). This bit is especially important to check in Ineks if you have implemented and tuned the parameters of your homework in your own PC, as then in Ineks your code might run very slowly or very swiftly due to the change of the screen's refresh rate.