CPSC 427, A1: Game Graphics Assignment

Due: 23:59 PM, Friday February 11, 2022

1 Introduction

The goal of this assignment is to introduce you to basic graphics programming. You will experiment with rendering, shaders, and event-driven frameworks in general. It builds upon the tinyECS framework covered in the preparatory assignment, which you should at least partially solve before starting this one.

In the assignment you will implement a simple 2D game where the user controls a chicken flying in the sky. Can your chicken dodge the eagles that rush by? How many bugs will you eat? You will implement this game by building on top of an instructor-provided template, adding the required code.

The assignment includes both a required (80%) and a free-form component (20%). The goal of the latter is to let you experiment with computer graphics and have fun.

2 Template

The template code provides a starting base for your work. You will find comments throughout the files to help guide you in the right direction. The directory is structured as follows:

- The directory `src` contains all the header (`.hpp`) and source (`.cpp`) files used by the project. The entry point is located in `main.cpp` while most of the logic will be implemented in the world system (`world_system.cpp`) and physics system (`physics_system.cpp`).
- The `data` directory contains all audio files, meshes, textures, and shaders used in the code.
- The external dependencies are located in the `ext` subdirectory, which is referenced by the project files, it contains header files and precompiled libraries for:
  - `gl3w`: OpenGL function pointer loading (header-only)
  - `GLFW`: Cross-platform window and input

- `SDL/SDL_mixer`: Playing music and sounds
- `stb_image`: Image loading (header-only)
- `glm`: The GLM library provides vector and matrix operations as in GLSL
- `tinyECS`: A minimal entity component system library

2.1 Transformations and Rendering

The template uses OpenGL 3.3 with object transformation and projection matrices passed to the shaders. The projection matrix is set to orthographic with a view frustum of 900×600 that matches the window resolution (in `RenderSystem::draw()`). In order to ease the concatenation of multiple object transformations, such as scaling and translation, we provide the following functions (semantics resemble legacy OpenGL with glTranslate(), glRotate() etc.):

- `transform()`: The transformation is initialized to the identity
- `transform.rotate()`: Applies a rotation matrix to the current transform
- `transform.scale()`: Applies a scale matrix to the current transform
- `transform.translate()`: Applies a translation matrix to the current transform
- The sequence of transformations is stored as a 3×3 matrix that is then passed to the Vertex Shader and multiplied by the (orthographic) projection matrix.

Be careful about the order of transformations as they are being multiplied **before** being passed as uniform data to the shaders.

3 Required Work (80%)

1. Getting Started (10%, prereq: C++ Dev. Env. tutorial):

   (a) Download the assignment package from the course website and unzip the source template. It should match the structure specified in the Template section of this document. It also contains a video `a1_reference.mp4` demonstrating what your solution should look like once the required parts are completed.

   (b) Play the `a1_reference.mp4` to get a sense of what a possible assignment solution should look like.

   (c) The template is built using CMake, installed as detailed in the preliminary assignment. In the following, we will provide operating system-dependent instructions to install the additional dependencies of Assignment 1:

   Windows: it should be sufficient to open the repository folder (the one containing the CMakeLists.txt) with Visual Studio (you may have to install the VS CMake

extension) and pressing *Build*. For a manual CMake setup on Windows follow the instructions for Linux (below), omitting the installation of external dependencies.

Linux: please install `libglfw3-dev`, `libsdl2-dev` and `libsdl2-mixer-dev` using your package manager, such as `apt-get install <package name>`. Create an empty build directory named `build` in the `template` folder (`template/build`). You can configure the project using the CMake GUI or the command line. For the GUI, enter the assignment template folder (which contains a CMakeLists.txt file) as Source and the `build` folder as Build. Then, press *Configure*, and if the configuration if successful, press *Generate*. For the command line, cd inside the `build` directory and run:

```
cmake [path_of_assignment_template] -DCMAKE_BUILD_TYPE=[Debug|Release]
```

Now you can build the generated project using 'make' from the command line at the top-level directory, or using your favorite IDE. You will require a compiler that supports C++14.

Note: running CMake and building the project will copy files and data to the `build` folder. **Do not edit any files in the build folder**, only edit files in the `src` and `shader` folders (create new assets in the `data` folder).

The template provided should compile and run on Windows and Linux. Please contact the TAs if you have any challenges with installation on these platforms. If you have an Apple device you may want to work remotely on the department Linux machines (ask TAs for advice on how to connect).

(d) To verify that the installation was successful, compile and start the program from the command line by following the prior commands, or from your IDE. It should start an OpenGL window containing a rendered chicken and eagles. Make sure that your debugger works, as detailed in the previous assignment.

You are allowed to work on the assignment in any environment and OS but we expect your code to compile and run on the lab machines (Linux).

2. A Playable Game (40%, prereq: ECS lecture): You will find comments in the source files marked with `TODO A1` to help guide you in the right direction. For a basic version of the game make the following changes to the provided template:

(a) Game loop: The chicken is spawned at the games start in `WorldSystem::restart()`, and eagles are added periodically in the games loop `WorldSystem::step()` with random positions and constant velocity. Inspect these sections of code to understand the game state. It is your task to update the positions of all entities by their respective velocities in `PhysicsSystem::step()`. When implemented, eagles should move to the bottom of the screen while the chicken stays stationary with velocity $(0, 0)$.

(b) Chicken movement: pressing the Up/Down directional keys should make the chicken fly up and down and pressing the Left/Right directional keys should make it fly left and right; until the respective keys are released. The keyboard callback function is located in `WorldSystem::on_key()`. Use it to keep track of the state of the keys. You can then use it to directly modify the chicken's position or to update its velocity. The chicken's position and velocity is stored in a Motion data structure that is retrieved with `registry.motions.get(player_chicken)`. It is the same motion data that you have modified in task (a).

(c) Eating bugs: Similar to the eagle, insert bugs at random in `WorldSystem::step()`. A bug is instantiated with `createBug()` defined in `world_init.hpp`, give them half the speed of the eagle. Once this is working, modify the code to spawn bugs and eagles to the top of the screen, outside of the players eye. The eagles are dangerous for the chicken, while the bugs can be eaten by the chicken in order to obtain points.

(d) Rotation (prereq: rendering lecture): Provide mouse control for rotating the chicken. Change the movement of the chicken to follow its forward direction (e.g. if the chicken is looking down right then forward should make it move down right as well). The mouse position should rotate the chicken and the left/right keys move the chicken along the direction it is aligned with. You can obtain the mouse position in the `WorldSystem::on_mouse_move()` in window-coordinates, relative to the top-left of the screen. You can then calculate the rotation angle with respect to the chicken's default facing direction (positive X axis) using the `atan2(y,x)` function, which then can be used to update the chicken's orientation. Orientation is stored in the Motion structure alongside position and velocity. In order to render the correctly oriented chicken you will also need to modify `RenderSystem::drawTexturedMesh()` and issue the `transform.rotate()` command in the correct order.

(e) Collisions: While the basic collision code is already implemented in `PhysicsSystem::step()`, you need to properly handle the interactions between entities in `WorldSystem::handle_collisions()`. Upon collision with an eagle, modify the chicken's motion to be upside-down and make the chicken fall downwards.

3. OpenGL and Shaders (30%, prereq: rendering lecture): It is most efficient to load all required resources (mesh, shader, and textures) at once in the beginning of the game, and to keep these separate from the dynamic game logic. Inspect how the lower part of `components.hpp` declares all the available resources. You will have to return here for adding new assets. The actual resources, such as the mesh and texture file names, are described in the `render_system.hpp` and loaded in the `initializeGlGeometryBuffers()` function in `render_system_init.cpp`. Locate and inspect the mesh and texture loading functions. Note also that the mesh is constructed/loaded differently: The chicken has a more complex geometry and each vertex has its own color, while the eagle and bugs are 'faked' using a texture, which is applied on a quad (two triangles).

   Inspect the `createBug()` and `createEagle()` functions, they are similar. Compare

these to the `createChicken()` function. Analyze how their `renderRequests` tie back to the different textures and shaders.

Rendering is initiated by `RenderSystem::draw()`, which in turn calls `drawTextured Mesh()` on all the entities in the game with a `RenderRequest` component. Based on the resources specified in `RenderRequest`, different shaders are called and different arguments are passed to the shaders. Otherwise, the OpenGL draw commands are the same for all entities.

(a) Collision animations: Trigger the following animations upon chicken collisions:

i. Eagle: If a collision with an eagle occurs the chicken's alive state is changed. Add code that changes its color. Study `drawTexturedMesh()` to understand how the `color` variable is simply another component and how it is passed to the vertex shader variable `fcolor`. Open `shader/chicken.fs.glsl` to see how it's being used to modify the final chicken color. Then modify the `color` variable to make the chicken red after a collision, and switch back to its original color on reset.

ii. Bug: Whenever a chicken eats a bug, the score should be updated in the window title and the chicken should temporarily light up. The chicken is drawn lit up in the `shader/chicken.fs.glsl` shader based on its state which is passed as a uniform variable from `drawTexturedMesh()`.

Proceed in two steps:

i. Create a new struct called `LightUp` in `components.hpp` and add an instance to the chicken entity upon chicken-bug collision. Equip `LightUp` with a timer. You can follow a similar implementation for the chicken's death with the `DeathTimer` struct. Remember to add the new class to the ECS registry as well as to count down all new timers.

ii. Pass the correct state to the shader in `drawTexturedMesh()` based on whether it has a `LightUp` component and change the light color from white to yellow inside the shader.

(b) The wind effect demonstrated in the example video is achieved using a second-pass shader. Two-pass rendering is done by first rendering the screen to an off-screen texture (see `RenderSystem::draw()`). Then, in the second pass a fragment shader is used to apply additional effects to each pixel of the texture obtained from the first pass (`RenderSystem::drawToScreen()`). This is achieved by rendering a full-screen geometry in a similar fashion to how the eagles and bugs are rendered. The two-pass rendering code is provided in `RenderSystem::drawToScreen()`. Your job for this part is to modify the wind fragment shader `shaders/wind.fs.glsl` for the wind distortion and color shift. Note that you do not need to match the solution video exactly.

- Hints for the distortion (`distort`): think about the translation, what if the offset value is not uniform at all pixel locations but is varying like a wave

function? What if this wave function is varying based on time? Another helpful piece of information is that the input and output values of `distort` are in $[0, 1]$, so you should set the offset values to the right scale.

- Hints for potential seam artifacts: your distort function may output values outside of $[0, 1]$, leading to wrapping artifacts at the screen border. Reduce the deformation effect towards the screen boundary to ensure it stays in range.
- Hint for the color shift (`color_shift`): check the function `fade_color` in the same file. You want to shift the world slightly to yellow.

4 Creative Part(20%)

The required code changes described so far will let you earn up to 80% of the grade. To earn the remaining 20% you will need to make the game more appealing by implementing one advanced feature. You may also gain bonus points when exceeding our expectations. **Marks for the advanced features will be granted only if both they and all basic features are fully implemented and functional.** Advanced feature suggestions:

- Give the chicken momentum such that it continues moving even when no keys are pressed, while slowly slowing down due to the drag in sky. Search online for plausible models of air friction. Implement a suitable one and explain your choice in the README file.
- Diversify the types of obstacles floating in the air. Add two new objects with new visuals and new behaviors, such as eagles flying in randomized arcs and a vortex that pushes all chicken, bugs and eagles towards it; be creative!
- Add multiple chickens and let the user control the chicken that is closest to the mouse cursor, forcing the player to multitask. The ECS system should ease the addition of multiple chicken entities.

Use your imagination to make other additions than the ones listed above, however, please make sure you focus on tasks involving OpenGL, ECS, and animation knowledge.

To support both basic and advanced visualization and control features, **you need to add a toggle option where the user switches between the two modes** by pushing the 'a' and 'b' keys ('a' for advanced mode and 'b' for basic mode; either at startup or during the game).

Document all the features you add in the README.md file you submit with the assignment. Advice: implement and test all the required tasks first before starting the free-form part.

To get full credit you should add at least one of the advanced features above and make it fully functional **and** free from bugs. The grading of additional bonuses, features, and the size of bonuses will be at the marker's discretion. A bonus is given for solutions that go beyond the

suggestions listed above. **Multiple partially implemented features will not receive full credit.**

5 Hand-in Instructions

1. Create a folder called `cs-427` with subfolder `a1`. Copy all your source files and the CMakeLists.txt as present in the template to this folder (same folder structure as the assignment; the TA should be able to run CMake and compile). Double check that you include the `shader` folder. Exclude all generated files, such as /build, .vs, /out, and the example videos! These would consume a lot of space on our server.

2. In addition, create a README.md file (Markdown language as used on GitHub) that includes your name, CWL, student number, and any information you would like to pass on to the marker.

3. The assignment should be handed in with the exact command `handin cs-427 a1` on a department machine (use SSH to do this remotely).

   This will handin your entire a1 directory tree by making a copy of your a1 directory, and deleting all subdirectories. If you want to know more about this handin command, use: man handin. You can also use the web interface on your myCS page to upload the assignment.

Recall, do not publish your solution on github or any other place. Neither during the course nor after; both is considered cheating.