CPSC 427 - A1: Video Game Programming
Game Graphics Assignment

September 12, 2021

1 Introduction

The goal of this assignment is to introduce you to basic graphics interface programming. You will experiment with rendering, shaders, and event-driven frameworks in general. It build 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 salmon swimming upstream. Can your salmon dodge the turtles that rush by? How many fish 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 modern OpenGL with object transformation and projection matrices passed to the shaders. The projection matrix is set to orthographic with a view frustum of 800×1200 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 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 how your solution should look like once the required parts are completed. The package can be downloaded from the course website.
   (b) Play the `a1_reference.mp4` to get a sense of what a possible assignment solution should look like.
   (c) The template is build 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.

   On 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 to hit build. For a manual CMake setup, create the same build folder structure and execute the same cmake command as for the other operating systems.

For OSX users, install dependencies using Homebrew (`https://brew.sh/`):

```
brew install pkg-config
brew install glfw3
brew install sdl2
brew install sdl2_mixer
```

And for MacPorts (`https://www.macports.org/`):

```
port install pkgconfig
port install glfw
port install libsdl2
port install libsdl2_mixer
```

For Linux users, please install `libglfw3-dev`, `libsdl2-dev` and `libsdl2-mixer-dev` using your package manager, such as `apt-get install <package name>`.

Create an empty directory as the build directory, which we assume is named `build`, you could place it in `template/build`. Note that running CMake and building the project will copy files and data to this folder. **Do not edit any files in the build folder** since they can be overwritten during the build process. Only edit files in the `src` and `shader` folders (create new assets in `data`).

You can configure the project using CMake GUI or the command line. For the GUI, enter the assignment template folder (which should contain a CMakeLists.txt file) as Source and the `build` folder as the Build. Then, press configure, and if the configuration if successful, press generate. For the command line, cd inside the `build` and run:

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

Now you can build the generated project using make or your favorite IDE. CMake can be configured to output project folders for Visual Studio, Xcode and others. You require a compiler that supports C++14.

(d) To verify that the installation was successful, compile and start the program in your IDE. It should start an OpenGL window with a salmon and turtles appearing. Make sure that your debugger works, as detailed in the previous assignment.

2. A playable game (40%, prereq ECS lecture): Running the provided game template should now display a colorful salmon on the left and turtles spawn on the right. Make the following changes to make it playable. You will find comments throughout the files to help guide you in the right direction and entry points for this assignment are marked with `TODO A1`.

   (a) Game loop: The salmon is spawned at the game start in `WorldSystem::restart()` and turtles are added periodically in the game loop `WorldSystem::step()`, with random position and constant velocity. Inspect these code parts 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, the turtles should move to the left while the salmon stays stationary with velocity $(0, 0)$.

(b) Salmon movement: pressing the Up/Down directional keys should make the salmon swim up and down and pressing the Left/Right directional keys should make it swim 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 salmon position or to update its velocity. The samlmon's position and velocity is stored in a Motion data structure that is retrieved with `registry.motions.get(player_salmon)`. It is the same motion data that you have modified in task (a).

(c) Fish prey: Likewise to the turtles, insert additional fish at random in `World System::step()`. A fish is instanciated with `createFish()` defined in `world_init.hpp`, give them twice the speed of the turtles. Once this is working, modify the code to spawn fish and turtle to the right of the screen, outside of the players eye. The turtles are dangerous for the salmon, while the fish can be eaten by the salmon in order to obtain points.

(d) Rotation (prereq Rendering lecture): Provide mouse control for rotating the salmon, so that moving the mouse to the left/right rotates the salmon clockwise/counterclockwise. 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 calculate the rotation angle with respect to its default facing direction (positive X axis), which can then update the salmon orientation. Orientation is stored in the Motion structure alongside position and velocity. In order to render the correctly orientated salmon 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 `World System::handle_collisions()`. Upon collision with a turtle, modyify the salmon's montion to be upside-down and make the salmon sink 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` 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 salmon has a more complex geometry and each vertex has its own color, while the turtle and fish are 'faked' using a texture, which is applied on a quad (two triangles).

Inspect the `createFish()` and `createTurtle()` functions, they are alike. Compare these to the `createSalmon()` function. Analyze how their `renderRequests` tie back to the different textures and shaders.

Rendering is initiated by `RenderSystem::draw()`, which in turn calls `drawTexturedMesh()` 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 salmon collisions:

  i. Turtle: If a collision with a turtle occurs the salmon's alive state is changed. Add code that changes its color. See `drawTexturedMesh()` to understand how the `color` variable is just another component and how it is passed to the vertex shader variable `fcolor`. Open `salmon.fs.glsl` to see how it's being used to modify the final salmon color. Then modify it to make the salmon red after a collision and switch back to the original color upon reset.

  ii. Fish: Whenever a Salmon eats a fish, the score is updated in the window title and the salmon should temporarily light up. The salmon is drawn lit up in the `shader/salmon.fs.glsl` shader based on its state which is passed as uniform from `drawTexturedMesh()`. Proceed in two steps:

   - Create a new struct called `LightUp` in `render_components.hpp` and add an instance to the salmon entity upon salmon-fish collision. Equip `LightUp` with a timer. You can follow a similar implementation to the salmon death with the `DeathTimer` struct. Remember to add the new class to the ECS registry as well well as to count down all new timers.
   - 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 underwater effect demonstrated in the example video is achieved using a second-pass shader. The two-pass rendering code is provided in `RenderSystem::drawToScreen()`. Your job of this part is to modify the water fragment shader, `shaders/water.fs.glsl`, for the underwater distortion and color shift. Note that you don not need to match the solution video exactly.

Hints for the distortion part (`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 $[0, 1]$, you should set the offset values to the right scale.

Hits 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 boundary or down-scale your deformation output to ensure it stays in range.

Hint for the color shift (`color_shift`): check the function `fade_color` in the same file. You may want to shift the underwater world slightly to blue.

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 turtles and fish are rendered.

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% to make the game more appealing by implementing one advanced feature. You can 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:

(a) Give the salmon momentum such that it continues moving even when no keys are pressed, while slowing down slowly due to the drag in water. Search online for plausible models of air/water friction. Implement a suitable one explain your choice in the README file.

(b) Diversify the types of obstacles floating down the river. Add two new object with new visuals and a new behavior, such as fish swimming in randomized arcs and a vortex that pushes all salmon, fish and turtles towards it; be creative!

(c) Add multiple salmons and let the user control that salmon that is closest to the mouse courser, forcing the player to multitask. The ECS system should ease the addition of multiple salmon 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 examples 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 and CMake files as present in the template to this folder (same folder structure). Double check that you include the `shader` folder. Excluded all generated files, such as /build, .vs, /out! 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, 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`

   This will handin your entire a1 directory tree. 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.