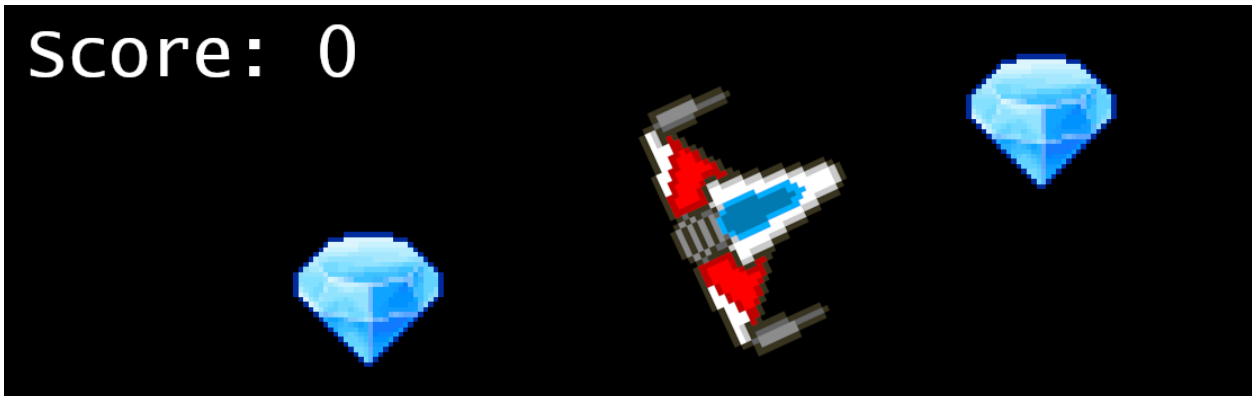
CPSC 453 Assignment 2 Transformations and Multiple Objects [*]

Fall 2021, University of Calgary



1 Overview and Objectives

The purpose of the second assignment in CPSC 453 is to provide you the opportunity to work with transformations, and a bit of the shader pipeline, by creating a very simple game. A modified version of the source code will be provided for you. It will contain all of the code necessary to render the textured quad that makes up the player character, but you will have to supply all of the transformations and game logic. As before, if you prefer, you may create your own template instead.

[*]Certain assignment specification taken from previous offerings of CPSC 453 by Sonny Chan, and Faramarz Samavati and a previous assignment write-up by John Hall. Please notify drlarsen@ucalgary.ca of typos or mistakes.

In addition to familiarizing you with how to draw multiple objects in a single scene, this assignment will be the first one in which you will need to touch the OpenGL shaders. This is because, in order to achieve full marks, all of your transformations will be implemented via the vertex shader rather than modifying the vertices directly in your C++ code. That is, to achieve full marks, all vertices in the C++ code will remain unchanged for the entirety of the program's run, including after the user specifies movements, and the vertices will be updated with these transformations only in the vertex shader (e.g. "test.vert" in the boilerplate code). For more information on this, you may read ahead to Part IV of this assignment.

2 Due Date

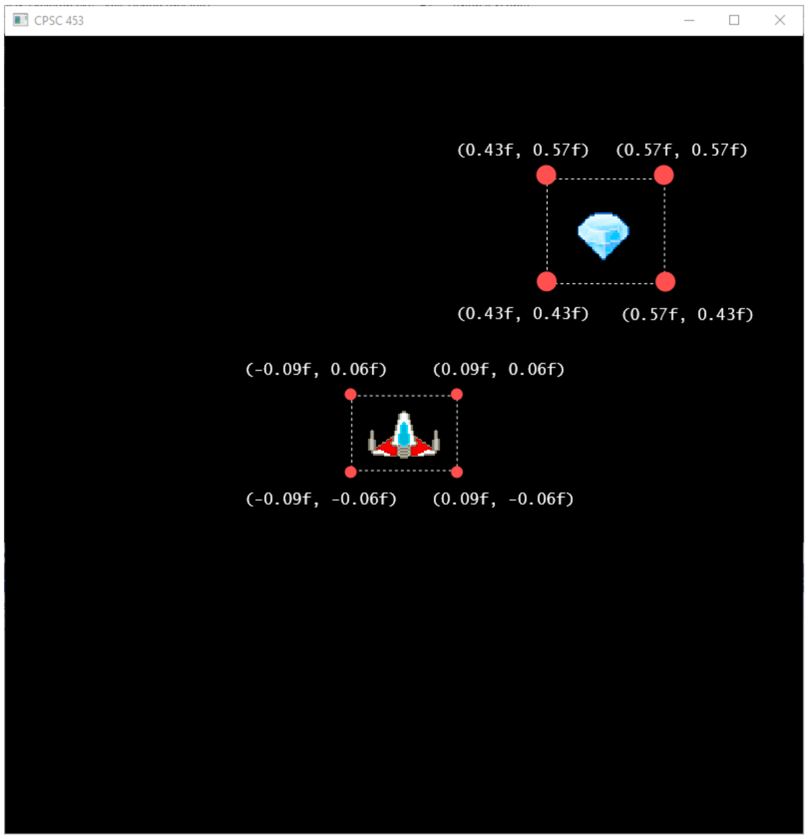
October 17th at 11:59 PM

3 Programming Assignment

There are a total of four parts to this programming assignment, all of which add up to make a single application with a single scene, the game. This assignment is worth a total of 20 points, with the point distribution as indicated in each part, and an additional possibility of earning a maximum of 5 "bonus" points, by completing one of the two bonus designations. Since the bonus is a binary state, it will only be awarded to submissions that do an exemplary job of meeting the requirements.

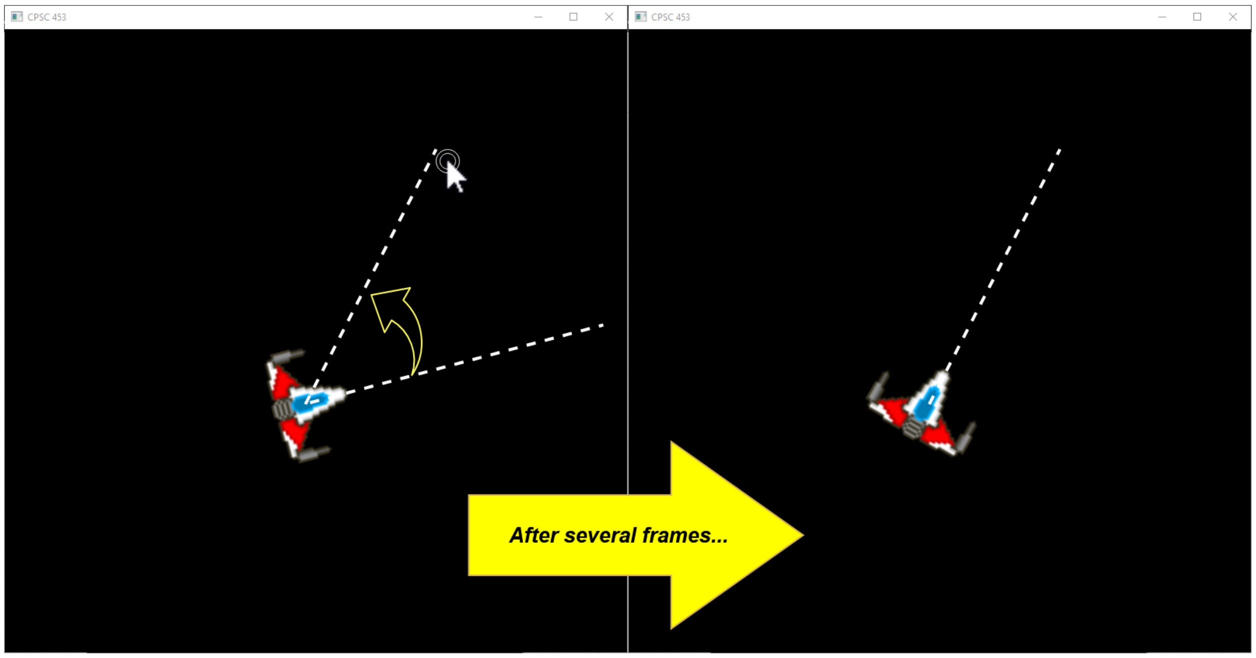
3.1 Part I: Displaying Multiple Objects with Images (5 points)

Create a program that reads the digital image files for the game objects and displays them on screen, within the bounds of the screen and with the correct aspect ratio. Code has already been provided to do this for the player character; learn from this existing code to do it as well for one of the other game objects. For now, you can focus on just getting the player and one other object to both render on the screen at the same time. That also means that, at this stage, you can choose to hard-code in the other object's location by setting its vertex coordinates away from the player's. However, at some point, all transformations must be done in the vertex shader, so if you want to start off by doing the transformations this way, you can do so here.

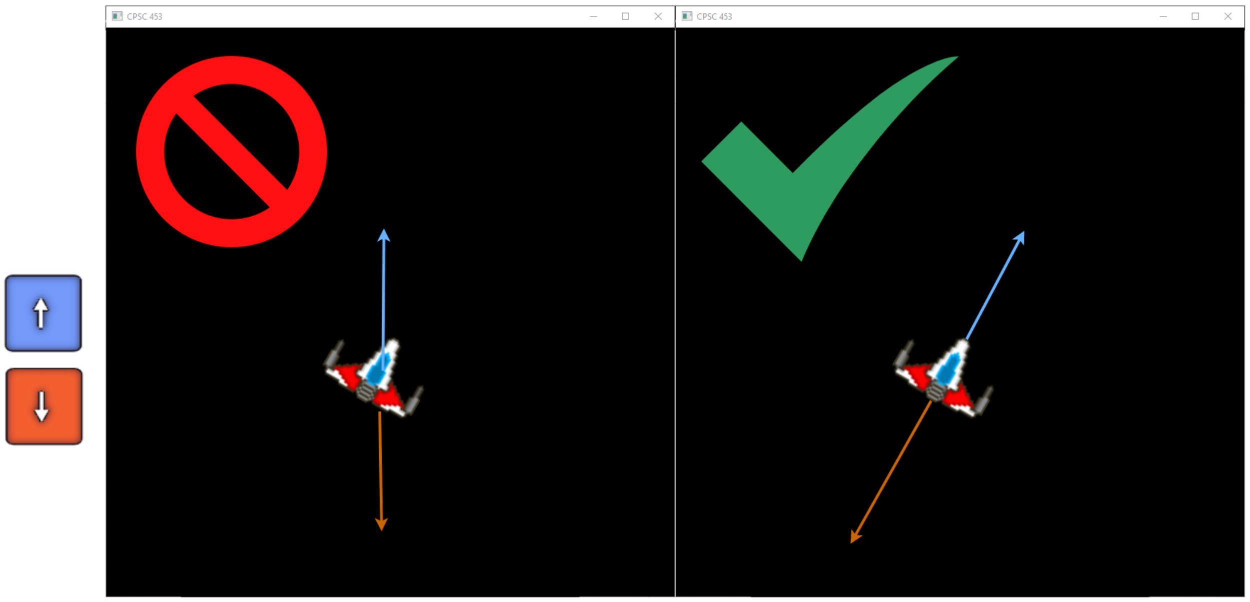


3.2 Part II: Moving the Player Character (5 points)

Add in the ability to move the player character around the screen. When the player clicks on the screen, the character should rotate so that they face the location of the click. This rotation will be done by rotating the quad on which the player is drawn. Make sure that the player rotates around its own centre, rather than around the centre of the screen.

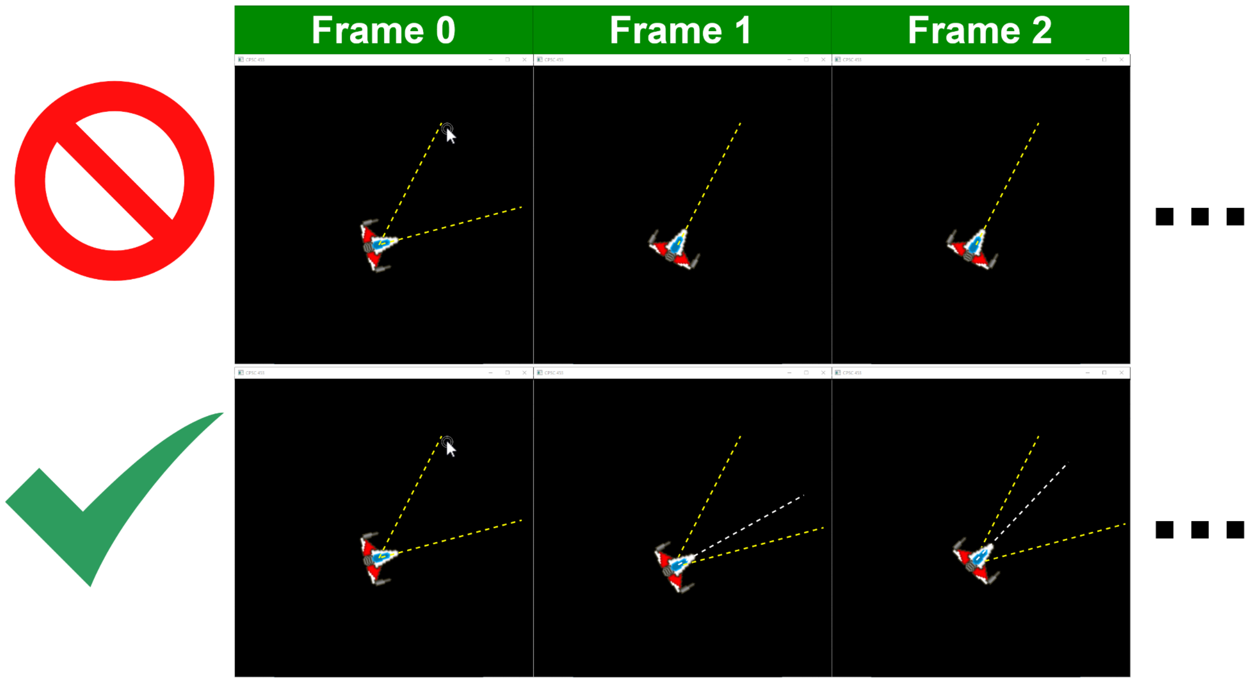


Additionally, allow the player to move forwards and backwards, in the direction they are facing (rather than just along the screen's horizontal or vertical axis), using keyboard input, e.g. the W and S keys or the UP and DOWN keys.



Once again, if you want, you can start off by implementing all these transformations in your C++ side, or you can get a head start on Part IV by implementing them via the vertex shader.

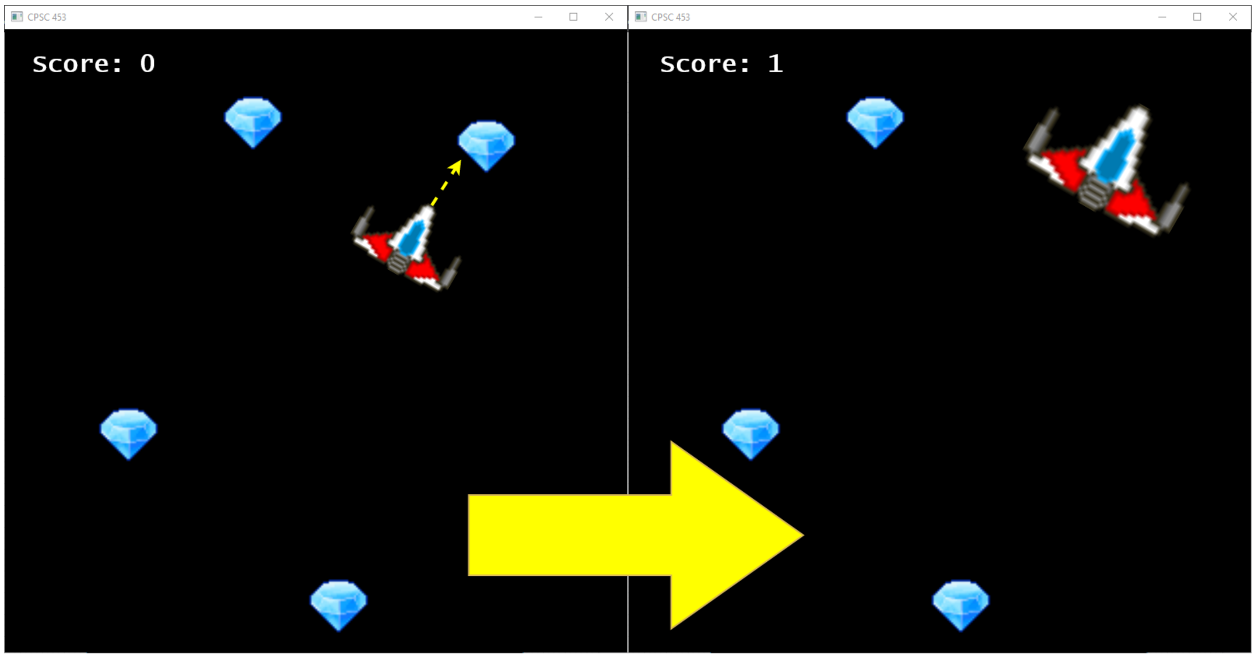
The rotation and translation of the player should both be done smoothly, i.e. it should look continuous rather than instantaneous. This means that, for example, if you need to rotate the player character 45 degrees for it to face the location of the latest mouse click, rather than rotating the player 45 degrees over the course of a single frame, it should take the player multiple frames to reach the target rotation, e.g. rotating one or two degrees per frame.



For full marks, it is critical that GLFW is used for all of the player's input, in the forms of mouse clicks and keystrokes as described above. For example, taking player input via the terminal is not an acceptable alternative and will not result in full marks.

3.3 Part III: Game Logic (5 points)

Now that you can move your player, add several pick-ups (at least three) in different locations of the screen, away from the player. Make it so that when the player gets close enough to the objects, the objects "disappear" and the player character grows slightly in size. Unlike rotation and translation, the change in scale can be instantaneous rather than continuous if you prefer. Make sure the player character's size grows about its own centre, rather than the centre of the screen. Each time the player picks up a new object, update the "score" text that is being rendered with ImGui, and once the player has picked up all of the objects, display a congratulatory message to the player, telling them that they have won.



Choose a key on the keyboard that, when pressed, restarts the game. Restarting the game should reset all translations, scales, rotations, and text, and all the objects should be visible once more. This key press should, once again, be handled via GLFW, not an alternative e.g. the console.

As with all previous parts, you can keep all of your translations in C++ for this part, or you can get a head start on the final part of the assignment by already writing them in the vertex shader.

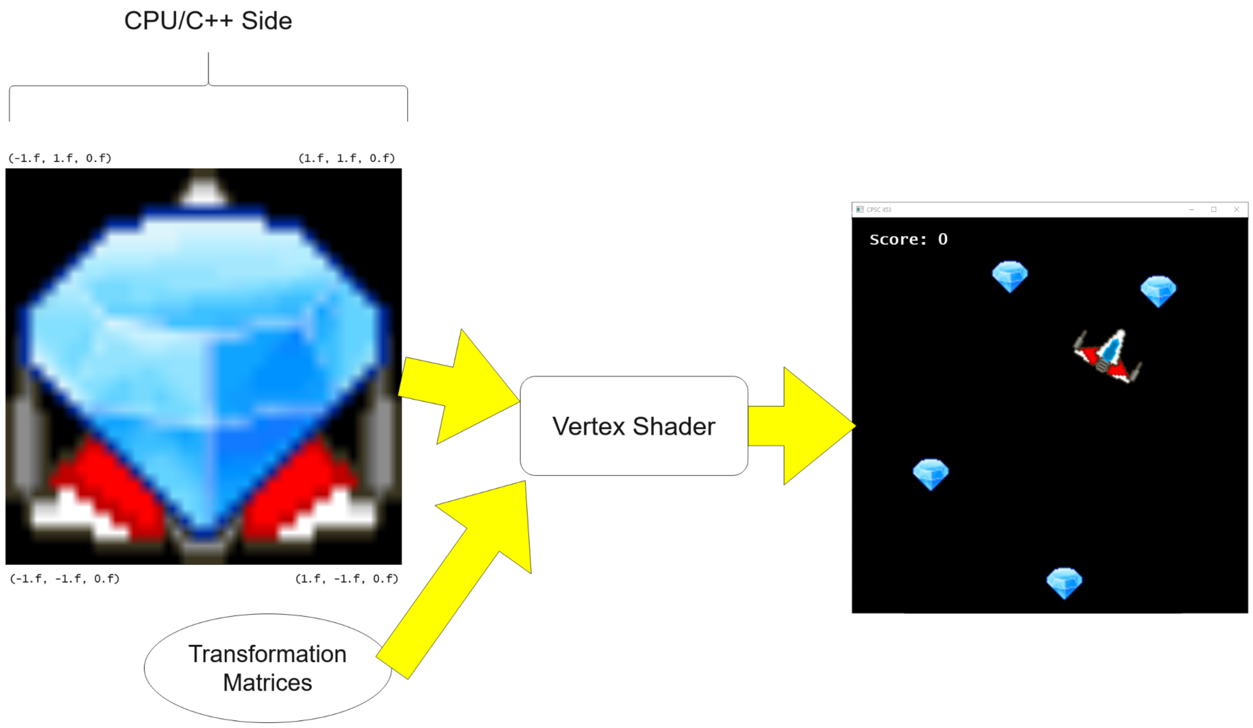
3.4 Part IV: Vertex Shader Transformations (5 points)

Up until this point in the assignment, you have had the option of only touching C++ code, without touching any of the GLSL shaders. Now, however, you must interact with the vertex shader.

Modify the creation of the CPU_Geometry for your objects so that each one's vertices are only initialized to the locations (-1, -1, 0), (1, -1, 0), (1, 1, 0), and (-1, 1, 0), i.e. the four corners of the window as represented in GL's coordinate system. That is, if you were to render the objects without performing any transformations in the vertex shader, each one would be stretched out to fill the entire window. No vertex positions other than the above four are acceptable for any of your objects, including the original player character.

Once you create the CPU_Geometry in this way, you may not touch the vertices of that CPU_Geometry in your C++ code anymore. That is, once you create your CPU_Geometry objects in the manner stated in the above paragraph, you cannot interact with their vertices ever again. This means that, in your render loop, as you log user input for rotations and translations, you cannot implement these transformations by altering the CPU_Geometry's "verts" attribute.

To change the placement, rotation, and scale of all your objects, you will keep track of their transformations in your C++ code via transformation matrices, and then you will pass these into the vertex shader as uniforms. Then, in the vertex shader, you will multiply each vertex's vec3 by the required transformation matrix. This product will then be passed into gl_Position as that vertex's transformed position.



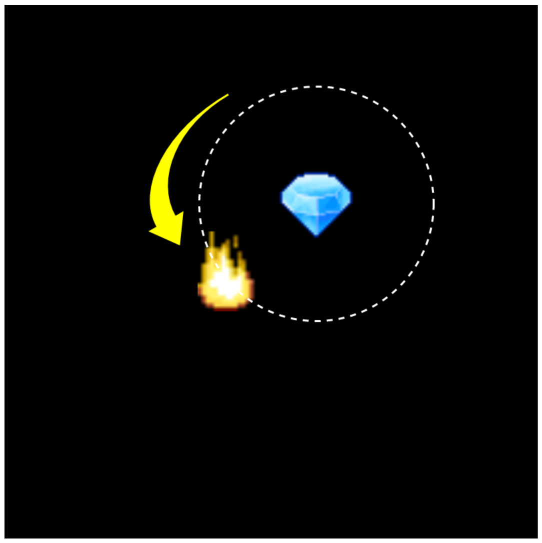
Note that, for this assignment, you only need to touch the vertex shader; the fragment shader will not require your attention until a future assignment, though you are of course welcome to investigate it and experiment with it on your own time.

3.5 Bonus Part I: Transformation Hierarchies (5 points)

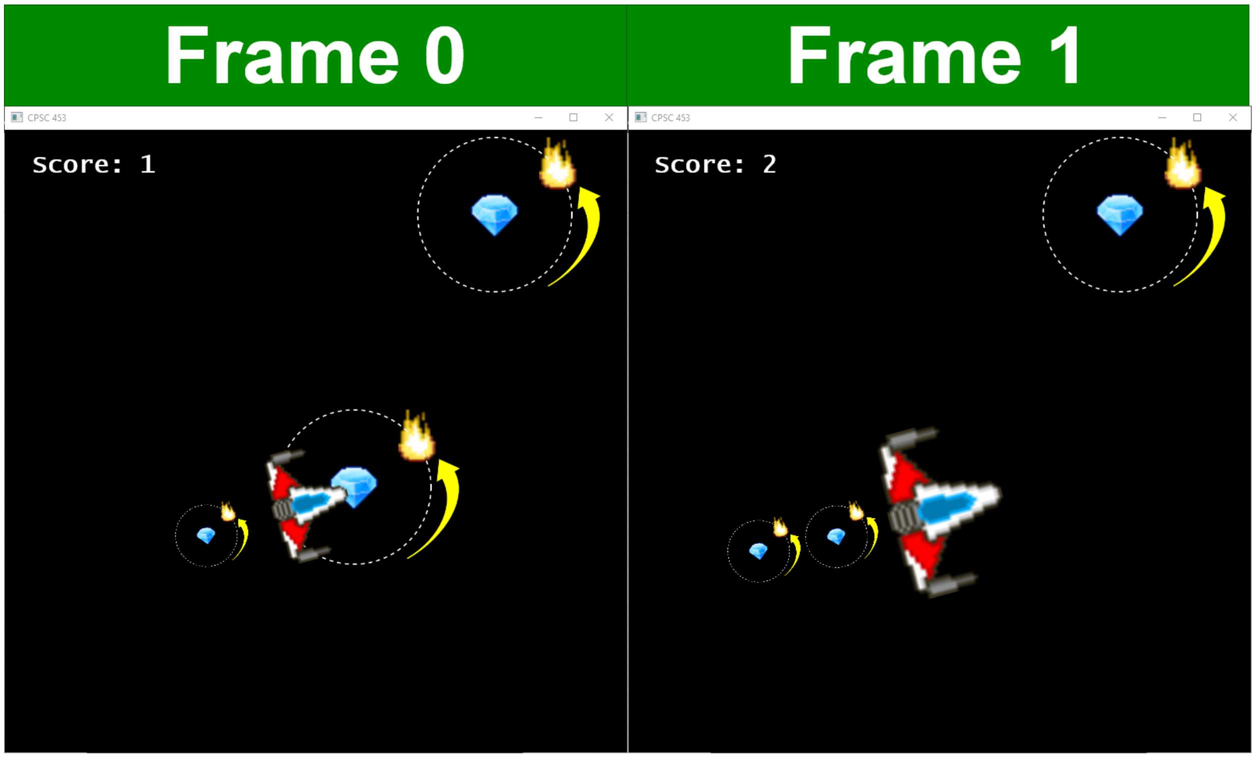
Note: *The 5 points awarded to this bonus cannot be combined with the 5 points from the other for a total of 10 bonus points. A maximum of 5 bonus points, from completing one of the bonuses in full, will be awarded.*

Update your game to include a system for hierarchical transformations, where some objects can be parented to others and then inherit their transformations.

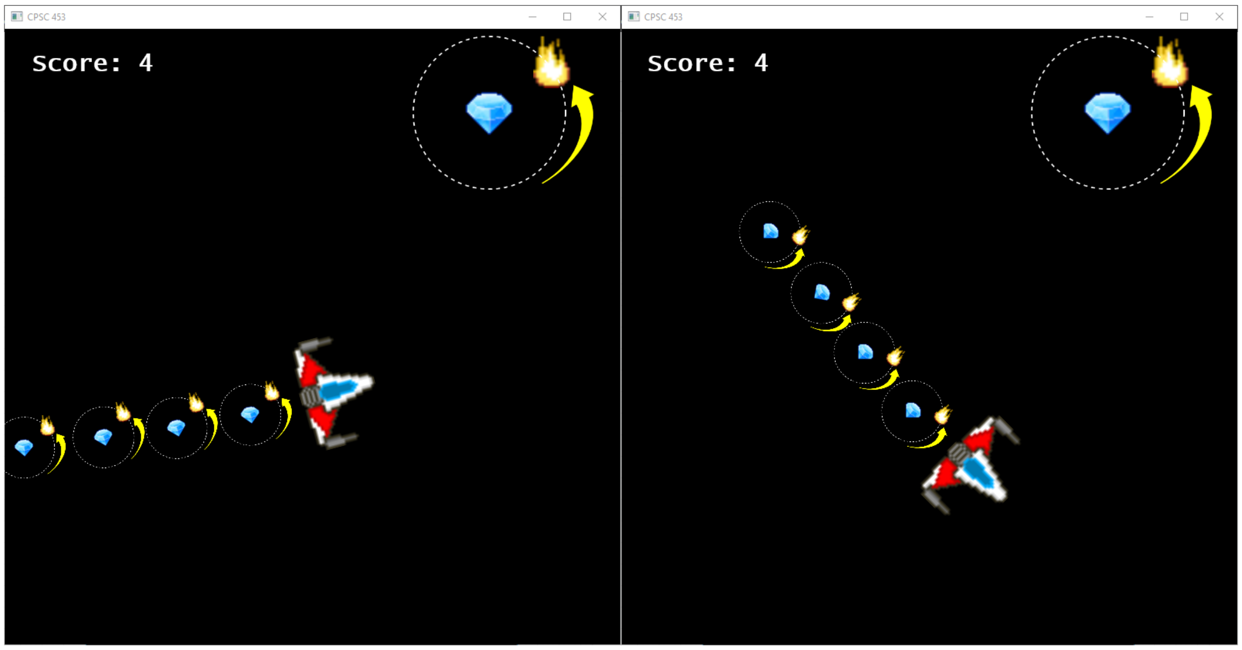
To start with, for each pick-up object, add an obstacle orbiting around it. We will provide the texture for this, as with all other objects, though you are free to use your own if you prefer. If the player hits this obstacle, the game should restart.



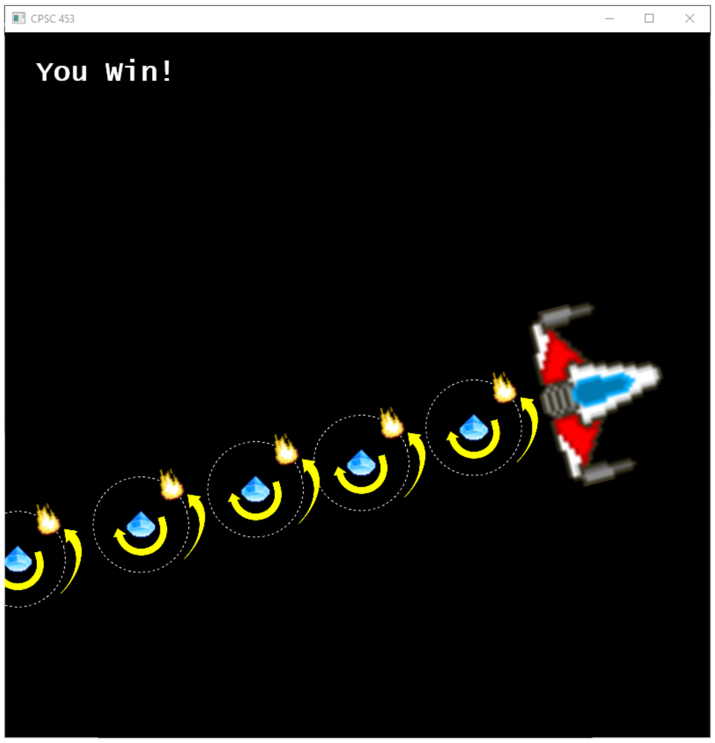
If the player collects pick-ups, then the pick-ups and their obstacles should be shrunk and placed in a row behind the player. This shift in location behind the player can be instantaneous rather than smooth/continuous.



Once the objects are behind the player, the player becomes their parent object. This means that, as the player rotates, the row of collected pick-ups should move so that they still form a line in the opposite direction of where the player is facing. All throughout this, the shrunken obstacles should still be orbiting their pick-up, though they no longer need to have any effect on the gameplay.



Once all pick-ups have been collected and the game has been won, have each pick-up start spinning around its own centre. The rotation of the pick-up should not affect the orbit of its obstacle.



3.6 Bonus Part II: Instancing (5 Points)

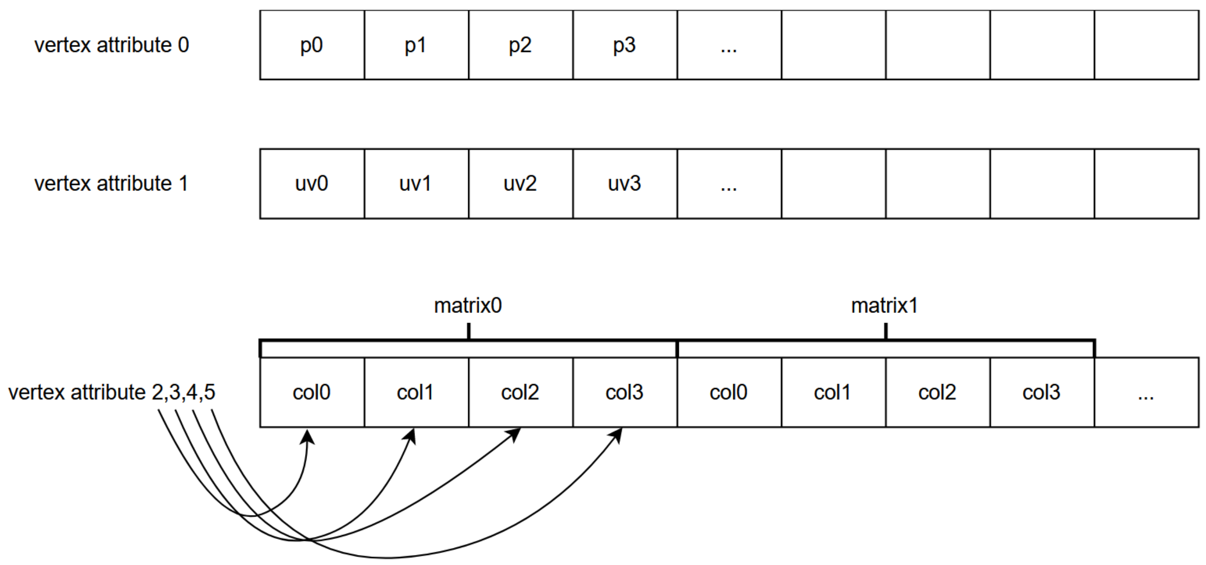
Note: *The 5 points awarded to this bonus cannot be combined with the 5 points from the other for a total of 10 bonus points. A maximum of 5 bonus points, from completing one of the bonuses in full, will be awarded.*

Up until now, we have allowed you to use a draw call for each item rendered on the screen. For this bonus, we want you to reorganize the way you draw things so that a single draw call is used per frame. Note that you must have completed all of the main components in order to be eligible for this bonus.

Graphics cards are becoming increasingly powerful. The sheer amount of vertices and pixels they can process in parallel is astounding. They are also becoming increasingly flexible in what they can do and how you can program them. However, this flexibility can come with a performance price. One of the most expensive parts of modern graphics APIs is context switching. Each draw call that you make requires the GPU to reason about how the render call must be performed which includes figuring out how the buffers (vertex data) is setup, which shader programs will be used, and which texture units will be used. The time it takes to set things up is expensive. As a result, using many draw calls per frame will significantly slow down your program. In this assignment, if you have done all of the required elements, the draw calls are all using the same vertex information. The only difference is in the uniforms. Each element that you draw has a different model matrix, and possibly a different texture.

OpenGL provides a way to issue a single draw call in situations such as this. The idea behind this technique is called "Instancing". Instances are objects that have similar geometry, but some details are different. They might be in a different location, have a different orientation and may have slightly different rendering details, such as colours or textures. glDrawArraysInstanced is a call that allows you to draw all of these instances with a single draw call. Using this call, you can easily draw tens of thousands of elements per frame.

To accomplish this, you will no longer use uniforms for you model matrices. Instead you will create a buffer to hold all of the model matrices. Each frame, you will update the matrix for all of the objects and upload this buffer to the GPU. You must also create a buffer that contains the appropriate texture ID for each instance and upload it to the GPU. Note that the model matrix contains 16 floating point values, which larger than the per-vertex data you have been using thus far. This requires you to use multiple vertex attributes per model matrix and you must also inform the GPU that this data is per-instance, instead of per-vertex. Figure X shows how the memory layout and vertex attributes is different in the case of a buffer containing matrices. To use the model matrix as a full matrix in the shader you must use multiple glVertexAttribPointer calls and multiple glVertexAttribDivisor function calls. Doing so will modify the rate at which the vertex attributes advance through the matrix buffer.



We have purposefully left out some of the information you will need to accomplish this from our tutorials and this description. You are free to use external thirdparty references for learning how to do this, but cite them in your README. An example reference is the excellent Learn OpenGL site which has an entire article devoted to Instanced rendering. If you have done this correctly you should be able to easily draw thousands or tens of thousands of items on the screen per frame.

Compilation of links for the above:

1. https://learnopengl.com/Advanced-OpenGL/Instancing
2. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawArraysInstanced.xhtml
3. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glVertexAttribDivisor.xhtml
4. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glVertexAttribPointer.xhtml

4 Submission

We encourage you to learn the course material by discussing concepts with your peers or studying other sources of information. However, all work you submit for this assignment must be your own, or explicitly provided to you for this assignment. Submitting source code you did not author yourself is plagiarism! If you wish to use other template or support code for this assignment, please obtain permission from the instructors first. Cite any sources of code you used to a large extent for inspiration, but did not copy, in completing this assignment.

Please upload your source file(s) to the appropriate drop box on the course Desire2Learn site. Include a “readme” text file that briefly explains the keyboard controls for operating your program, the platform and compiler (OS and version) you built your submission on, and specific instructions for compiling your program if needed. In general, the onus is on you to ensure that your submission runs on your TA’s grading environment for your platform! It is recommended that you submit a test assignment to ensure it works on the environment used by your TA for grading. Your TAs are happy to work with you to ensure that your submissions run in their environment before the due date of the assignment. Broken submissions may be returned for repair and may not be accepted if the problem is severe. Ensure that you upload any supporting files (e.g. makefiles, project files, shaders, data files) needed to compile and run your program. Your program must also conform to the OpenGL 3.2+ Core Profile, meaning that you should not be using any functions deprecated in the OpenGL API, to receive credit for this part of the assignment. We highly recommend using the official OpenGL 4 reference pages as your definitive guide, located at: `https://www.opengl.org/sdk/docs/man/`.