CSC 406 – Computer Graphics

Programming Assignment 02, Fall 2024

Wednesday, October 1st, 2024

Due date: Sunday, October 13th.

1 What this Assignment is About

1.1 Objectives

There are multiple objectives for this assignment:

- Develop more complex graphic objects;
- Implement some simple animations.

This is an individual assignment. You are of course encouraged to discuss it with other students but I expect you complete it by yourself.

1.2 Handouts

The handout for this assignment are this document (as a pdf file).

2 What to Do, Part I: Graphic Object as a Group

2.1 Re-use your portrait class

You are going to re-use the portait class you developed in the previous assignment. From this point onward, I am going to call it the `Portrait` class. Whatever name you gave to your class is fine and should be used instead in your code.

If you don't complete the first programming assignment, it's fine if you borrow that of one of your classmates, as long as you properly identifie the source.

The first thing that you should do is a bit or refactoring of your `Portrait` class:

- (if you didn't already have one) A constructor that lets you specify an origin, orientation, and scale for your portrait. Ideally, the origin should be about the center of the portrait;
- Refactor your `Portrait` class as the subclass of the `GraphicObject2D` class we worked on last week
- *Setter* and *getter* methods to get and set the position, orientation and scale of a given `GraphicObject2D` (because all instance variables defined in a class should be private).

- Make sure that your portrait class has a public `const` function named `draw`, with no argument, that overrides the pure virtual function defined in the parent class to draw the portrait.
- Update the `Portrait` class to work in "world units". You should only have to modify the dimensions of the different parts of your portrait so that it now gets drawn in "world units" (say, either centimeters or inches) rather than in pixels, as we saw and did in class. Everything should get scaled automatically if the window is resized.

This shouldn't take much time at all.

2.2 The `ComplexGraphicObject2D` graphic class

An `ComplexGraphicObject2D` object is a `GraphicObject2D` made up of a number of "parts" that are themselves `GraphicObject2D` objects. The `ComplexGraphicObject2D` has its own origin, angle, and scale. The position, orientation, and scale of the different parts of a `ComplexGraphicObject2D` are defined relative to the `ComplexGraphicObject2D` they belong to (in the parlance of game engines such as Unity, we would say that the parts are "parented" to the `ComplexGraphicObject2D`.

2.2.1 Drawing

The `ComplexGraphicObject2D` class won't be an abstract class since it can implement its `draw` function: It simply calls the `draw` function of its parts. It is probably a good idea to declare this function *virtual*, since a subclass may need to override the function.

2.2.2 Implementation details

We have decided to use smart pointers instead of raw pointers this semester. In particular, a `ComplexGraphicObject2D` will store a list[1] of smart pointers to its parts.

Report 1. *I leave it up to you whether these smart pointers should be shared pointers or unique pointers, but this will have to be discussed and justified in the report. Similarly, the way parts are added to a complex graphic object (Only at construction time? Through a dedicated function?) is left up to you.*

2.3 The `PortraitWheel` graphic class

You are going to implement one particular subclass of `ComplexGraphicObject2D`. This class will be named `PortraitWheel`. A `PortraitWheel` object is made up of a number of `Portrait` objects evenly distributed along a circle around the center of the group object. There are two kinds of `PortraitWheel` objects: "Heads on sticks" and "Heads on wheel," as shown in Figure 1.

[1]Here I mean list in the most generic form of the term. I don't mean at all that you should use the `std::list` data type.

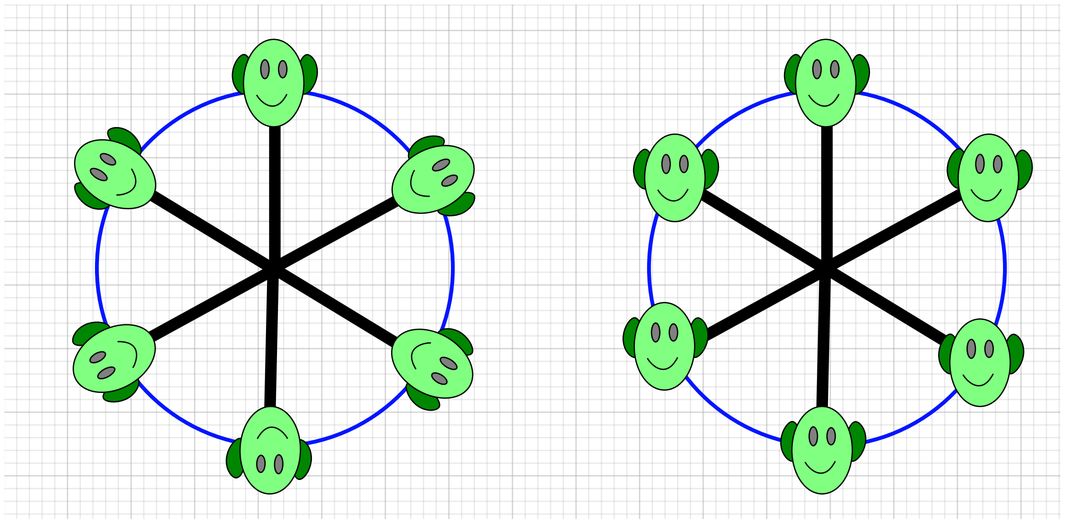

Figure 1: Left: “Heads on sticks” mode; Right: “Heads on wheel” mode.

A `PortraitWheel` object will be characterized by its type, position, orientation (they could spin), number of heads making up the group, and size of the “spokes” and heads (three sizes should be available: large, medium, small, each corresponding to an appropriate scale factor).

2.4 First, define some enum types and constants

First, define two enum types:

- `WheelType` admits two values (for the time being):
  - `HEADS_ON_STICKS`,
  - `HEADS_ON_WHEEL`.
- `WheelSize` admits three values (that you will map to a scale value):
  - `LARGE`,
  - `MEDIUM`,
  - `SMALL`.

Note: For this entire assignment, the elements of the API (identifiers, parameter signature, semantics of function calls) that are specified are *mandatory*. You are not free to change the name of a data type or of a function, nor to change the parameter signature (number, type, and order the parameters) of a function. Another student should be able to use your `PortraitWheel` *as is* and get virtually the same results as with their own class (except e.g. for small variations in scale).

2.5 Public API of the `PortraitWheel` class

The following public constructor must be implemented in your class:

- `PortraitWheel(WheelType type, WheelSize size, int num, float x, float y)`: This constructor creates a new `PortraitWheel` with the type, size, number of heads, and initial position specified.

Note that there is no need to implement a `draw` function, since the parent class already did that.

2.6 Program 2: Interactive creation of `PortraitWheel` objects

2.6.1 The program

All the source and header files of your program should be placed in a folder named `Program1`. The user will be able to use the keyboard to select modes of operation and the mouse pointer to create `PortraitWheel` objects at the location of mouse clicks.

2.6.2 Modes of operation

In its final version, your program will be able to switch between different modes of operation, when the user hits a key:

- Hitting the ‘s’ key will let the user switch to “head on stick” mode;
- Hitting the ‘w’ key will let the user switch to “heads on wheel” mode;
- Hitting the ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, or ‘9’ keys will let the user select the number of portraits on the next group to create.
- Hitting the ‘+’ or ‘-’ keys will increase or decrease the size of the next group to create. We are using only three set sizes, so it isn’t possible to get smaller than `SMALL` or larger than `LARGE`.
- As usual, hitting the escape key should cause the program to terminate.

A new `PortraitWheel` at the current mode settings should be created at the location of a mouse click.

2.6.3 Extra credit (8 points)

If the user hits the space key, this should toggle on/off an animation mode. When animation is on, then the `PortraitWheel` objects should spin at a fixed speed of 60 degree per second (a full revolution in 6 s).

3 Part II: Simple Rigid Animation

3.1 The program(s)

For all the versions that you implement, all the source and header files of your program should be placed in a folder named after the version (`Version1`, `Version2`, etc.). All these version folders should be placed in a folder named `Program2`.

3.2 General objective

We are going to animate a small cart moving along a curvy road.

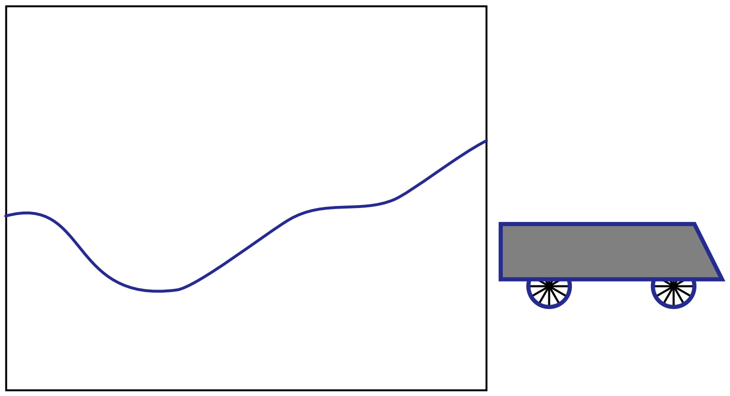

Figure 2: Example of a curvy road and the cart we want to see travel along that road.

You can draw the cart any way you want, preferably better than the one I drew in Figure 2, but there are two aspects of the cart I must insist on:

1. The cart must have clearly distinguishable front and back sides.
2. The cart's wheels must have spokes or markers that will make it clearly visible when the wheels spin.
3. The cart will obviously be implemented as as subclass of `ComplexGraphicObject2D`.

In what follows, I suggest development steps that let your progressively add all the features of the application. These are just suggestions. Fell free to develop the entire thing in one shot, but then don't come back to me to complain when you have a mess of incomplete parts that don't work well together.

3.3 Step 1: The world and the road

Needless to say, we will be drawing and animating things in world units. Just pick units and dimensions that make sense at the scale of a car (meters or feet seem indicated). The origin of the world will in the lower left corner of the window, with the Y axis pointing up, as shown in

Figure 3. Set the dimensions of your world and cart so that the world's width is at least ten times that of the cart.

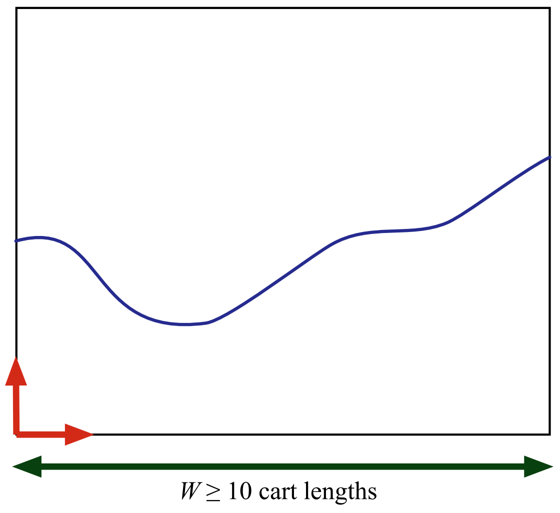

Figure 3: The world and the road that goes through it.

The road will be represented by an equation of the form $y = f(x)$. In the final version of the application you will need two different roads, and hence two different equations. In this assignment you will implement your road as a `Road` object.

What differentiates two `Road` objects is the equation of the road that they represent. We can't make the road's equation a simple instance variable (of `String` type) that would get initialized when the `Road` object is created and would need to be interpreted to determine the value of $y = f(x)$ for a give value of x. Or rather, we could, but this would become its own assignment that has nothing to do with computer graphics[2]. This means that the road's equation has to be hard-coded into a public method the `Road` class. There are two ways of doing this:

- Composition (*has a*): We hard-code n different road equations `private float roadFunc1(float x)`, `private float roadFunc2(float x)`, etc. Different roads are then distinguished by an index set in the constructor. The public `roadFunc` method simply consists in a switch statement that invokes the proper private `roadFuncXY` method based on value of the road object's instance variable storing the road's type.

- Inheritance (*is a*): We implement an abstract parent class `Road`, with an abstract public method `public abstract float roadFunc(float x) = 0;` and each road is implemented as its own class implementing its own `roadFunc` method with a hard-coded expression.

[2]Prof. Lamagna teaches a graduate course, CSC 550: Computer Algebra, that deals with this topic, in case you are interested. Alternatively, we could communicate with the Mathematica kernel to evaluate a function specified as a spring. This is one of the things that I cover in my course CSC450: Scientific Computing.

C++ Program 1. *Implement the* `Road` *class. Create a road when you initialize your application, and draw them in your display callback function. Just to make sure that everything works fine, create and draw two roads.*

Report 2. *You will definitely need to explain in your report (or in your Doxygen comments, as discusssed in Section 5) the choices you made for the different instance variables of your class, and how you draw your road.*

3.4 Extra credit (2 pts)

Regardless of whether you chose composition or inheritance, implement your `Road` class(es) so that it is only possible to instantiate one object with a given equation.

3.5 Step 2: Implement the `Cart` class

Obviously the cart has to be defined in "world units." As you design your `Cart` class, keep in mind that–later on—the cart object will have to move across the world, along the road, that its orientation will change as it follows the road, and that the cart's wheels will spin as the road travels. In this version, we only care about the position of the cart, not its orientation (and even less that of the wheels).

C++ Program 2. *For this version of the application, record the x coordinate of a mouse click, and display the cart object at the proper location (but fixed horizontal orientation) for that x value along the road.*

Report 3. *Obviously here there is going to be some interaction between the application (more specifically the mouse handle function), the* `Road` *class, and the* `Cart` *class to pass around the proper x coordinate and matching y coordinate on the road corresponding to the location of a mouse clicks. You will definitely need to explain in your report (or in your Doxygen comments) the choices you made for the different instance variables of your* `Cart` *class.*

3.6 Step 3: Add proper orientation to the `Cart` class (Version 1)

C++ Program 3. *For this version of the application, record the x coordinate of a mouse click, and display the cart object at the proper location and orientation for that x value along the road.*

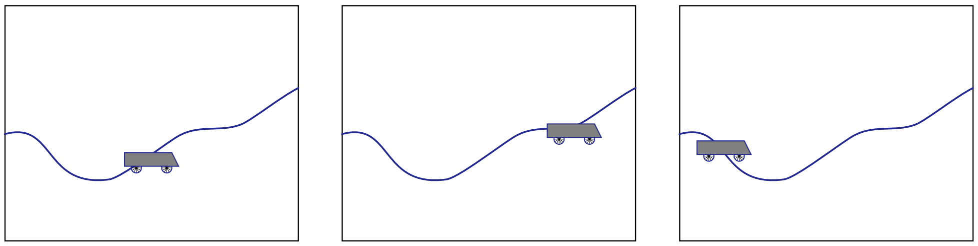

Figure 4: The cart along the road, but at constant orientation.

Report 4. *Discuss how you compute the orientation to apply to the cart. Make sure to explain your design decisions.*

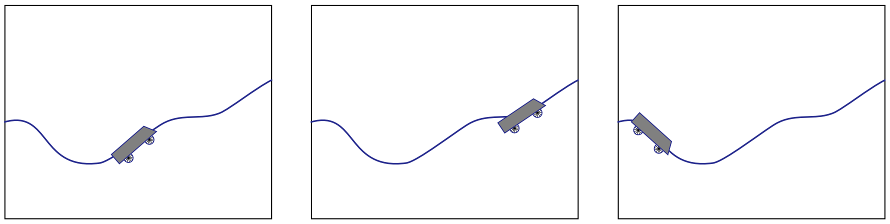

Figure 5: The cart along the road, with the proper orientation.

3.7 Step 4: Animate the cart object along the road (Version 2)

Position initially the cart at the left end of your world. When the user hits the space key, animate your cart by moving at constant speed along the x axis. When the cart arrives at the right edge of the world, invert the orientation of the cart, and flip its drawing so that the "front" of the cart points in the direction of motion.

Use the ',' and '.' keys to decrease and increase the speed of the cart along the x axis.

Report 5. *This step should be fairly straightforward, but report on any tricky aspects you may have encountered and how you handled them.*

As we can see on Figure 6, the x component of the speed vector (indicated by the red arrow) is constant. Where the slope of the road is steeper, whether going upward or downward (left and right cases), the speed also has an important vertical component (the green arrow), resulting in a longer total speed vector (orange arrow) that is not necessarily tangential to the road. When the slope is weaker (central case), the vertical component of speed is small, resulting in a slower total speed of the cart.

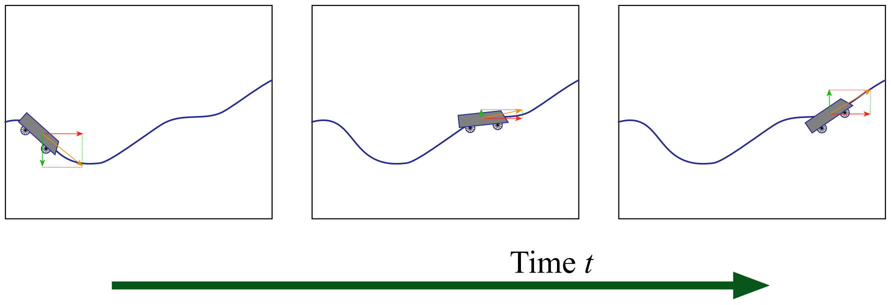

Figure 6: The cart along the road, with constant speed along the x direction.

3.8 Step 5: Animate the cart object along the road (Version 3

One problem with the previous version is that you cart ends up moving a lot faster when the road's slope in the xy plane increases. What we want to ensure in this version is that the speed we control is the true tangential speed along the road, and not only its component along the x direction. As in Version 2, use the ',' and '.' keys to decrease and increase the speed of the cart along the road.

Report 6. *Make sure to explain your speed computations in the report.*

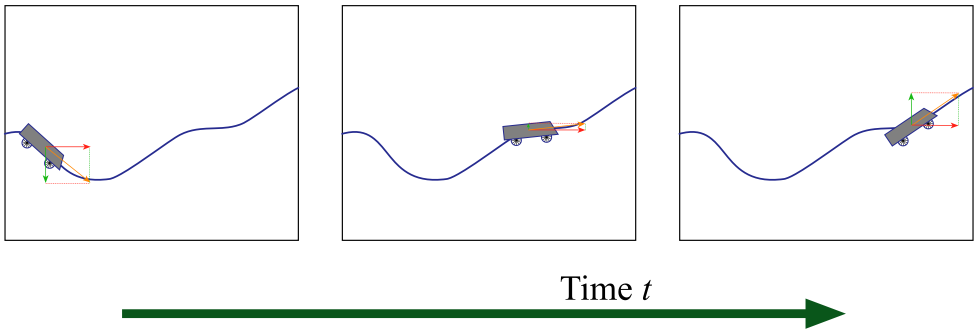

Figure 7: The cart along the road, with constant tangential speed.

By contrast to what we observed in Figure 6, the total speed vector of the cart is of constant magnitude in all cases of Figure 8, and the speed vector is tangent to the road. This means that the horizontal speed changes depending on the slope: It is smaller when the slope is steep and larger when the slope is gentle.

4 What to Do, Part II: Added Complexity

The "added complexity" items listed here are to be applied to the Version 2b code base. The complete, final Version 3 that you submit should have all these part implemented to get full credit.

4.1 Step 6: Spin the wheels (Version 4)

As a cart moves along the road, its wheels should spin. Not just spin at some arbitrary angular speed, but at the angular speed that corresponds to the tangential speed of your car along the road.

C++ Program 4. *Add spinning wheel animation to the application.*

Report 7. *Simply explain the mathematics behind the animation.*

4.2 Extra Credit

4.2.1 Extra Credit 1: Multiple instances (6 pts)

If your classes are correctly implemented, this step should hardly take any time at all to complete, just a few lines in the main class.

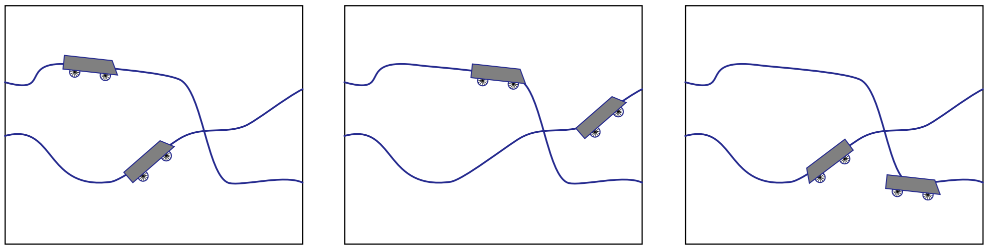

Figure 8: Multiple roads, each traveled by a single cart.

C++ Program 5. *Your application should now display three different roads, each with a different equation and a cart on that road. You should still be able to control individually the speed of an individual cart along its road, but first you need to select the cart by hitting its index on the keyboard.*

Report 8. *Simply explain the mathematics behind the animation.*

4.3 Extra Credit 2: Cart selection (5 pts)

Allow the user to select (through a keystroke, say `'1'`, `'2'`, `'3'`) a cart to control (by hitting the `','` and `'.'` keys, as discussed in Subsections 3.7 and 3.8.

4.4 Extra credit 3: Ball pickup (5 pts)

Add to your little cart scene a generator of small disks/balloons/coconuts/etc. that continuously fall from the top part of the world. A falling object that hits a cart is "picked up" by the cart and should disappear from the world (3 of the 7 points are for the quality of the code implementing the solution: how you maintain your list of falling objects and handling of collision detection).

4.5 Extra credit 4: Score (2 pts)

Display the "score" of the cart(s) at the top of the window.

5 What to Do, Part III: Doxygen documentation

Starting with this assignment, you will have to write Doxygen-style comments for all your methods and static variables. I will not always remind you of that in future assignments, but we will always attribute a part of the grade to your Doxygen comments.

You will need to provide Doxygen comments for each of your classes (what the class is about), and all of the methods, instance and class variables within that class. There are may features in Doxygen. For the time being (and probably, until the end of the semester), you only need to look at the fields `@author`, `@param`, and `@return`. You will also need to provide a summary for each class and function.

What this new Doxygen requirement means is that your list of function and variables and what they do is now going to be directly included in your code. You don't need to write this in the report anymore. The report will consists only of discussion of the problem and of the way you solved it.

6 What to Submit

6.1 Report

You should provide a report in the PDF file format, explaining the whatever design decision you may have had to make to complete the assignment, and current limitations of your program.

6.2 Code

Provide one separate folder for each version that you implemented.

6.3 zip archive

Place your report and source folders in a folder named `Prog 02`, zip that folder (zip only, not rar), and submit this archive on BrightSpace.

6.4 Grading

If you submit code that could not possibly compile (contains obvious syntax errors) your assignment will not be graded and you will get a grade of 0. For all others:

- Execution: 60 pts
  - Program 1: 20 pts
  - Program 2: 40 pts
    - Version 1: 10 pts
    - Version 2: 10 pts
    - Version 3: 10 pts
    - Version 3: 10 pts
- good design: 10 pts
- general quality of the code: 10 pts
- Doxygen documentation: 10 pts
  - Doxygen comments: 5 pts
  - Generated HTML documentation: 5 pts
- report: 10 pts