CSC 581 – Homework 1

Jayesh Bhagyesh Gajbhar (Unity ID – jgajbha)

The following is a team effort and the names of the contributors have been mentioned at the beginning of each file in the codebase. The modifications done by a contributor to another contributor's code, however, have not been annotated.
For example:

```
//
// Created by Jayesh Gajbhar on 8/28/24.
//
```

Section 1: Initial SDL 2 Project

For this section of the assignment, we utilized the code provided on moodle and made a few modifications to it.

We added custom libraries, a timer variable using the SDL_GetTicks() function and an fpsTimer which is part of the Timer class, which is a little buggy as of now and will be revised in future iterations in interest of modularity.

We have decided to follow the SOLID approach for this entire project. More information about the SOLID approach can be found here: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

This approach will help us achieve maximum modularity and ease of use in future implementations of the game engine. While not strictly adhering to all the SOLID principles, we try to implement it as much as possible (in interest of saving time). This pattern will be more apparent throughout this project as we progress further.

We have named the game engine Shade.

Section 2: Drawing Entities

Now that we have our base, we can implement drawing entities. In the current state, the Shade engine supports drawing only rectangles on screen.

We use Factory based coding systems to create entities. This helps ensure modularity in the code (source code : lib/objects/factory.cpp) . The factory's createRectangle() method returns a smart pointer to a Rectangle instance.

The Rectangle class is based on the Object class. The Object class uses the SDL_rect class.

The Object is described by 10 attributes, namely:

i. SDL_FRect rect : which defines the position as well as dimensions of the objects.
ii. SDL_Color color: which defines the color of the object.
iii. bool is_rigidbody: which defines whether the object is rigid (true) or not (false) [will be used in section 5]
iv. float mass: which defines the mass of the object [will be used in section 5]
v. float restitution: which defines the restitution constant of the object [will be used in section 5]
vi. float orientation: which defines the orientation of an object in the game environment [for future purpose]
vii. SDL_FPoint velocity: which defines the velocity of the object in a 2D coordinate system.
viii. float rotation: which defines the angular velocity of an object [for future purpose]
ix. SDL_FPoint acceleration: which defines the acceleration of the object in a 2D space.
x. float angular_acceleration: which defines the angular acceleration of the object [for future purpose]

Apart from these attributes, the Object class has the following member functions:

- void update() : to update movement of the object with respect to time.
- void scale() : to update scale of the objects with respect to the window proportions.

To draw the objects on screen we use a draw() method which is a part of the Rectangle class.

This method uses the SDL_SetRenderDrawColor() method to render the color of the objects, SDL_RenderDrawRectF() method to draw the rectangle outline on the screen, and, SDL_RenderFillRectF() to fill the rectangle with the scaled dimensions.

In my implementation, I have drawn 4 rectangles, which are as follows:

- The teal rectangle is controllable by Keyboard inputs. Use the WASD keys to move the teal rectangle.
- The yellow rectangle demonstrates the gravity capabilities of the Shade game engine.
- The red rectangle is a platform that moves from left to right at fixed intervals of time.

- The purple rectangle is an immovable, non-controllable, non-rigid object.
- All the objects in this demonstration have collisions implemented except for the purple rectangle.
- To resize the window, simply use the mouse pointer, and to change between scaling types, press ‘P'.

(The screenshots of the demo have been appended to the appendix)

Section 3: Applying Physics

Continuing with the theme of modularity, all the physics systems have been stored in the lib/core/physics folder.

The most important class in this file system is the Physics class, which contains the calculate() method which is overloaded based on the application. This class calculates and applies the appropriate forces on the object which it is attached to.

To implement gravity, a Gravity class has been created which inherits the calculate method from the Physics class. This applies a constant force to the object it is attached with. This force can be in any direction the developer chooses. This is because the gravity class can be initialized both an X and Y component. Here's the calculate method of the Gravity object:

```
void Gravity::calculate(Object &character) {
    character.acceleration.y = gravityY;
    character.acceleration.x = gravityX;
}
```

An example of applying gravity to an object is mentioned below:

```
gravity.calculate(*rectangles[1]);
```

In this case, the gravity object is initialized with a value of 100.0 in the Y direction (a constant downward force) and 0.0 in the X direction.

```
Gravity gravity(0, 100);
```

The screenshots of the demo have been attached in the appendix.

Section 4: Handling Inputs

For this part of the assignment, a keyMovement class has been created in the physics system. This class handles the movement of an object based on the direction input by the user.

To get the input from the user, the SDL_GetKeyboardState() method and SDL_SCANCODE variable have been used in the input.cpp file (located at lib/core).

The directions have been mapped to W for up, A for left, S for down, and D for right. The keyMovement class can be instantiated with any desired velocity for a keypress. For example,

```
KeyMovement key_movement(300, 300);
```

This initializes the speed of the movement to 300 in the X and Y directions respectively. This, multiplied with the direction of the movement gives us velocity in a 2D space. The following code will clarify this further:

```
if (state[SDL_SCANCODE_A]) {
    return {.x: -1, .y: 0};
}
if (state[SDL_SCANCODE_D]) {
    return {.x: 1, .y: 0};
}
if (state[SDL_SCANCODE_W]) {
    return {.x: 0, .y: -1};
}
if (state[SDL_SCANCODE_S]) {
    return {.x: 0, .y: 1};
}
return {.x: 0, .y: 0};
```

```
void KeyMovement::calculate(Object &character, SDL_FPoint direction) {
    if(direction.x == 0 && direction.y == 0) {
        character.velocity.x = 0;
        character.velocity.y = 0;
    }
    character.velocity.x = (direction.x * xVelocity);
    character.velocity.y = (direction.y * yVelocity);
}
```

The calculate method is inherited from the Physics class.

Section 5: Collision Detection

For purposes of this project, we have not only detected collision, but also handled it. The source code of collision handling is in the lib/core/physics folder.

To handle collisions, we first had to detect collision between two objects using the SDL_HasIntersectionF() method, which returns a Boolean true, if the objects are intersecting and false otherwise. We use SDL_HasIntersectionF() instead of SDL_HasIntersection() because we use FRect to define our objects which gives us more control over our objects as it uses float attributes rather than int attributes.

As stated earlier, the object has the attributes which define its mass, rigidity and restitution factor, which is useful here.

For handling collisions, we leverage an iterative approach to check for intersections between a main object, referred to as character, and a series of other objects within a vector. The function SDL_HasIntersectionF() is used to detect collisions, particularly when both objects are rigid bodies (i.e., when the objects' rigidity attribute is set to true)

Upon detecting a collision, the code calculates the collision normal, which is a vector pointing from the character to the colliding object. This normal vector is normalized to unit length. Additionally, the penetration depth, representing the amount by which the objects overlap, is calculated and used to adjust the positions of both objects, resolving the collision by moving them apart.

```
// Calculate collision normal
SDL_FPoint normal = {obj->rect.x - character.rect.x, obj->rect.y -
character.rect.y};
float length = std::sqrt(normal.x * normal.x + normal.y * normal.y);
normal.x /= length;
normal.y /= length;

// Calculate penetration depth
float overlapX = (character.rect.x + character.rect.w) - obj->rect.x;
float overlapY = (character.rect.y + character.rect.h) - obj->rect.y;
float penetrationDepth = std::min(overlapX, overlapY);
```

The code also calculates the relative velocity between the two objects and determines the velocity along the collision normal. The coefficient of restitution, which measures the elasticity of the collision, is used to calculate the impulse scalar. This impulse is then applied to both objects to adjust their velocities according to their masses.

Finally, the code applies damping to the velocities, reducing them by a factor of 0.7 to simulate the loss of energy typically observed in collisions. This value can be adjusted in code but in the current implementation, it is not as easy as changing the object attributes.

This detailed handling ensures that both the physical properties of the objects and the dynamics of their interactions are accurately modeled, enhancing the realism of the simulation.

This section was implemented with the help of the following resources:
https://www.metanetsoftware.com/technique/tutorialA.html
https://www.reddit.com/r/gamedev/comments/1wzhwv/simplest_2d_collision_solution/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1&utm_content=share_button
and github copilot (to correct errors in code)

Section 6: Scaling

To implement scaling in the game engine, a GameManager class has been created which handles whether the game objects need to be scaled with the window or not. It has a member variable

```
bool scaleWithScreenSize;
```

which, when set to true, scales the objects in the game proportionally to the game window.

Two modes of scaling have been implemented,

i. Constant size: shapes on the screen occupy the same number of pixels independent of window size.

   This is pretty straight forward and the objects stay the same size when the window is resized. This may result in objects going out of the viewport (shown in the appendix).

ii. Proportional: shapes on the screen increase and decrease size relative to window size.

   In this case, the scaleWithScreenSize variable is set to true and the objects in the game are scaled in proportion to the window size. This is achieved using the following code:

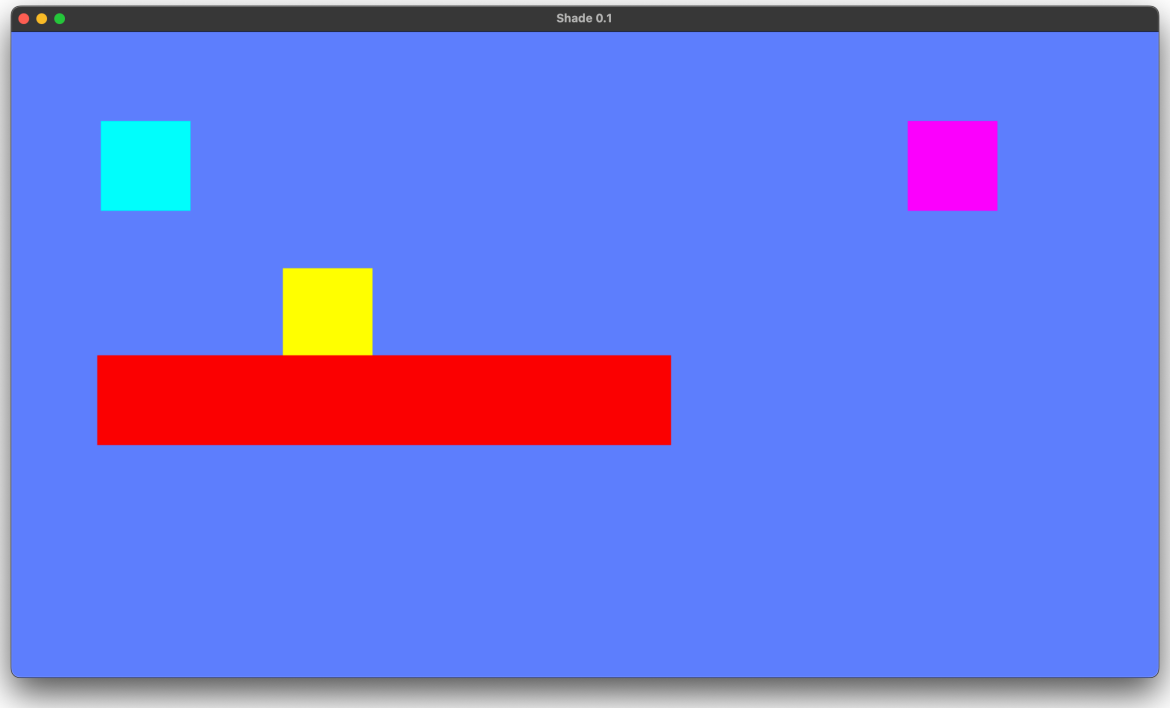
```
if (GameManager::getInstance()->scaleWithScreenSize) {
    int cwWidth, cwHeight;
    SDL_GetWindowSizeInPixels(app->window, &cwWidth, &cwHeight);
    SDL_FPoint scalingFactor = {
        static_cast<float>(cwWidth) / SCREEN_WIDTH,
static_cast<float>(cwHeight) / SCREEN_HEIGHT
    };
    float newX = r.x * scalingFactor.x;
    float newY = r.y * scalingFactor.y;
    float newWidth = r.w * scalingFactor.x;
    float newHeight = r.h * scalingFactor.y;
    r = {newX, newY, newWidth, newHeight};
}
```

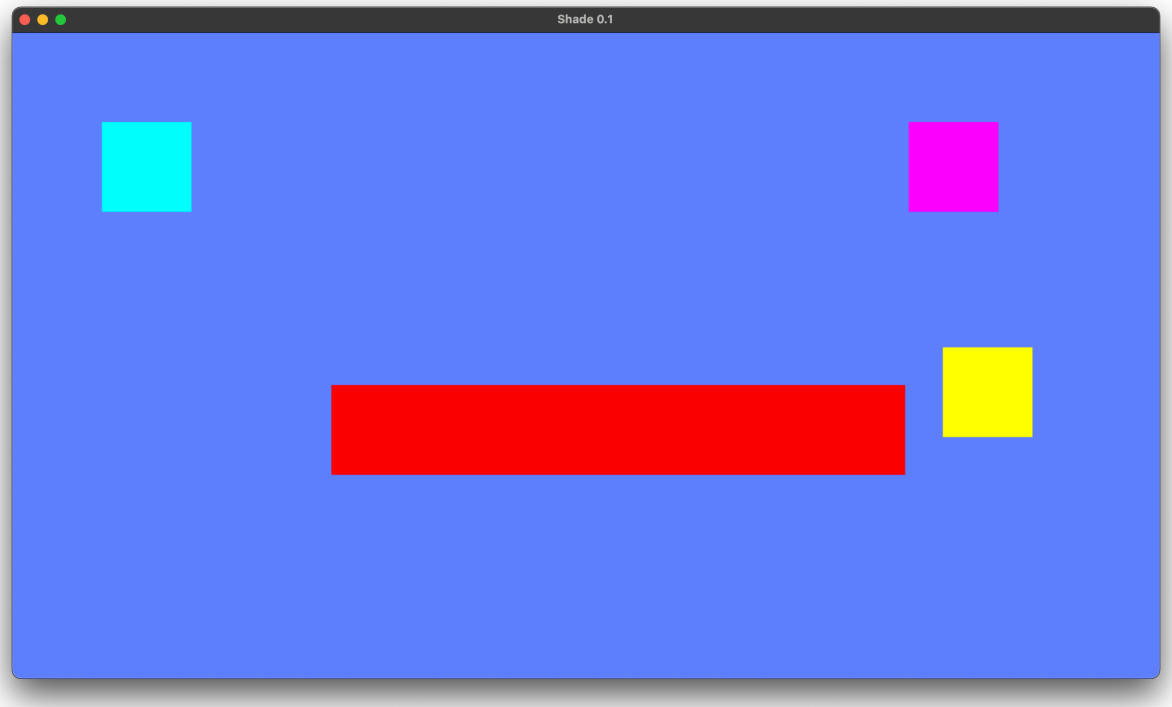
   which essentially calculates a scaling factor and multiplies it with the object to scale the object in accordance with the window size.

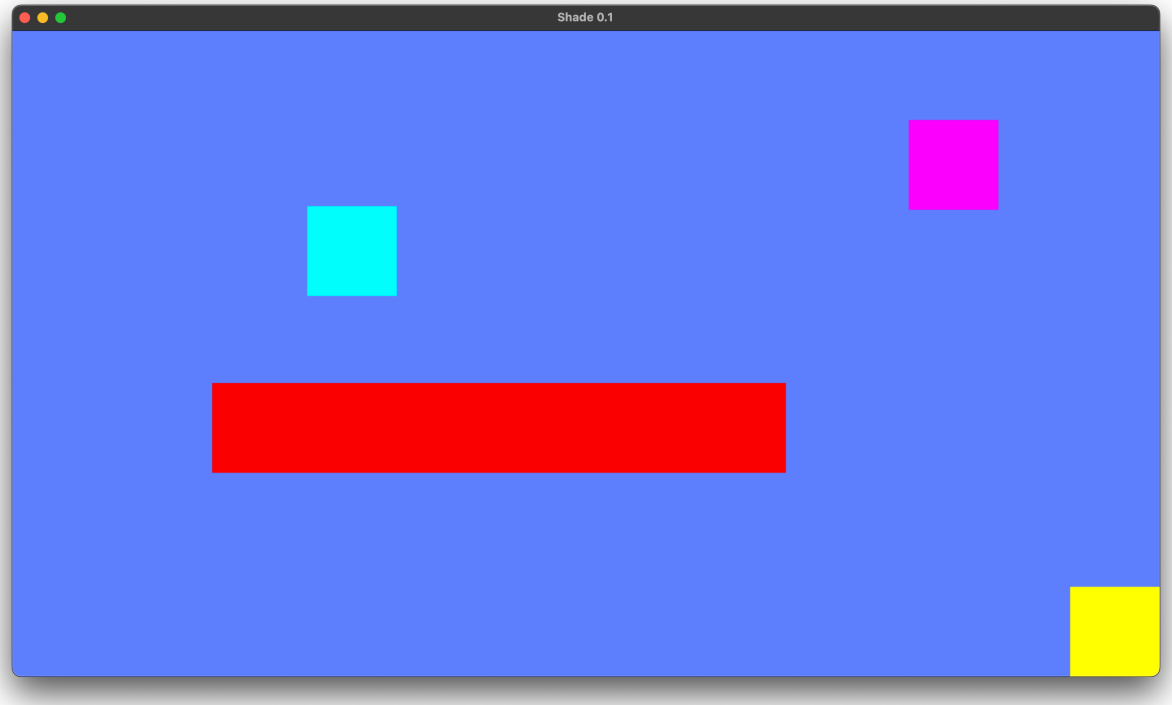
The scaling mode can be changed by pressing ‘P’ on the keyboard.

Finally, for my implementation, I have implemented all the sections mentioned in this assignment in one executable. The demo screenshots have been attached in the appendix and instructions on how to run the executable have been provided in the readme file.

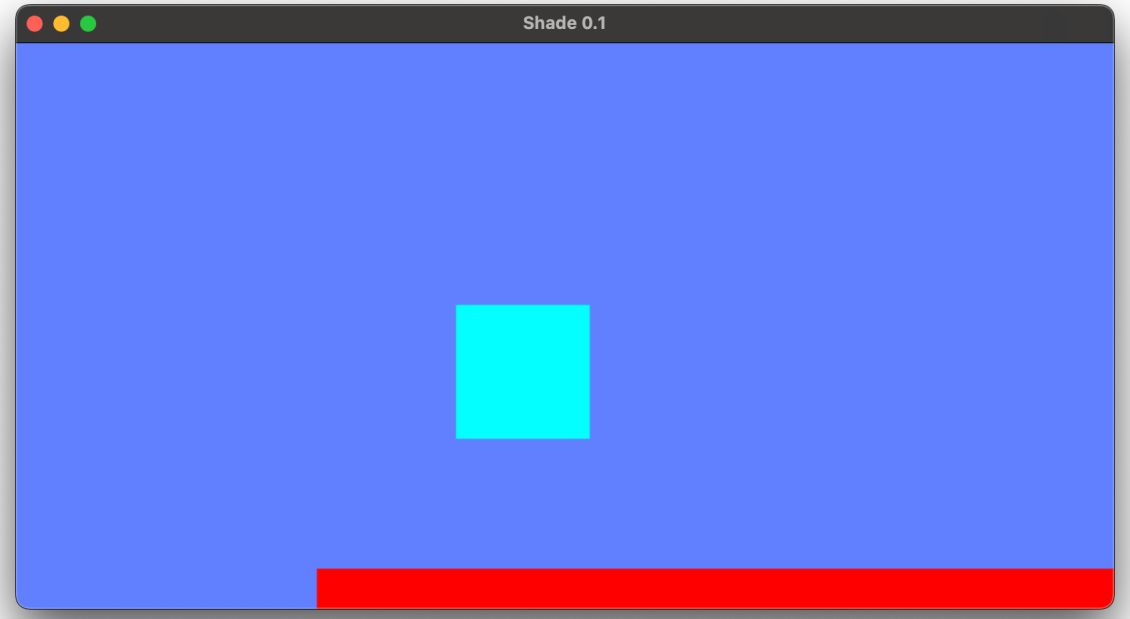
Appendix



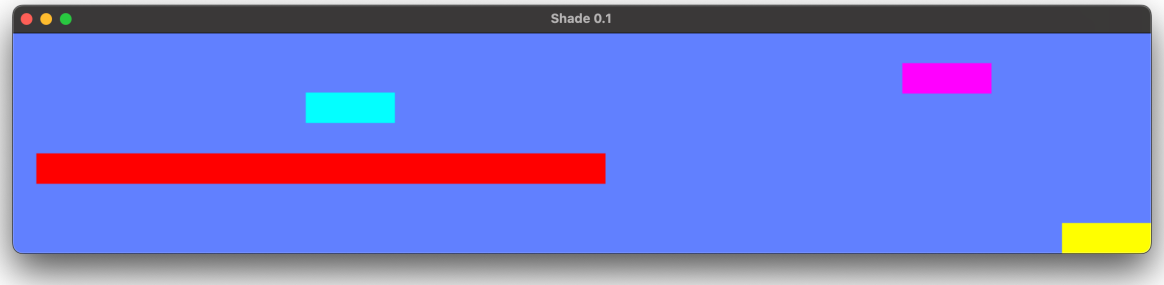




Above screenshots demonstrate the red rectangle moving in a continous pattern, the yellow rectangle being affected by gravity and collision physics, and the teal rectangle being controlled using the keyboard. The purple rectangle is non-rigid, non-controllable and non-interactable in the game environment.



Scaling with Constant size: shapes on the screen occupy the same number of pixels independent of window size.



Scaling - Proportional: shapes on the screen increase and decrease size relative to window size.