# CS 4730: Computer Game Design

SFX and Level Team: Environment Programming Part 1

***Overview***

For this assignment, you will implement all of the environmental interactions between objects necessary for your game's first two areas. This will lead to an alpha build in which your first two areas should be fully playable.

***IMPORTANT NOTES***

1) For this assignment, your larger group will submit a SINGLE BUILD of your game, with the additions of each of the three sub-teams incorporated. This will serve as an ***alpha release*** of your game.

2) In lab, you may have negotiated slightly different requirements than are specified in this document. If this applies to you, then work on the requirements you agreed to with your TA.

***PART 1: All Interactions***

Implement all interactions between objects in your game within the first two areas. This essentially means that all of the basic mechanics of your game will be completed once this assignment is completed. This means that, at a minimum, your first two areas should be fully playable. See the next page below for examples of how to easily hook up objects to interact with one another.

Your specific games may require different interactions, but most games will include the at least the following:

- Player interacting with platforms and other obstacles
- Player interacting with two special environmental objects per area (e.g., ice or similar)
- Player interacting with enemies and bosses
    - Your design team is implementing bosses this week, so be sure to coordinate with them.
- Player special items / moves interacting with platforms, enemies, bosses, etc.
- Player interacting with scene transitions points and transitioning to those scenes
- Player state being carried over consistently across scenes (e.g., coming back from battle scene in an RPG and health has been lowered and enemy now gone).
- Player losing the game and restarting scene or restarting whole game.
- Narrative text boxes appearing at appropriate times and disappearing at appropriate times.
- All controller input working as intended.
- Integrating UI elements to accurately reflect the state of the game (e.g., health bars)
- …more depending on your specific game.

You DO NOT have to develop out of game menus (such as the start screen, pause screen, etc.) for this assignment. Your engine team will do this in a future assignment.

***Part 2: Alpha Build***

This week, it should be possible to fully play (potentially with small bugs and lack of polish) the first two areas of your game. While you are responsible primarily for the environmental interactions, you should work with your team members to ensure everything gets merged together smoothly and correctly. Integrate these changes with everything your group has produced so far. This means that your demo this week will be fully playable for at least the first two areas.

***Turn In***

Submit a single zip of your game on Collab OR a link to a github repository with ALL of the additions from each sub-team incorporated.

****DON'T FORGET TO CHECK OUT THE EXAMPLE ON THE NEXT PAGE****

***Example***

Here is an example of how I chose to implement interactions between objects. *PLEASE BE AWARE that your other sub-teams (e.g., Engine and Collision System) may have altered their implementations. It is VERY IMPORTANT that you consult with your other team members and ask how relevant the code I'm providing here is for your game.

If your engine team implemented the Collision System as I did, then checking for collisions between objects is quite easy. Notice that most interactions between objects will be based on collisions, but certainly not all. You may have an object that only activates when the player gets nearby, or objects that operate on a timer, or particles that activate upon a button press, etc. To check for collisions though, I did the following:

```
    this->collisionSystem.watchForCollisions("Sayu", "Platform");
    this->collisionSystem.watchForCollisions("Enemy", "Weapon");
    this->collisionSystem.watchForCollisions("Sayu", "Enemy");
    this->collisionSystem.watchForCollisions("Blast", "Enemy");
```

The code above (which is taken from the MyGame class, which inherits from Game), simply asks the Collision System to look out for collisions between various types of objects of interest. For my simple demo, I was interested in four combinations [Sayu (the player) with platforms, Enemies with Weapons, Sayu with Enemies, and Blast (my character's melee attack) with Enemies].

The Collision System is configured to check all pairs of objects of the given types against one another (e.g., Sayu will be checked against all platforms in the scene). When the Collision System sees a collision occur, it lets both colliding objects know by calling the onCollision() method in each object (see image below). Thus, when I want an object to react to a collision, I simply implement onCollision() with the appropriate reaction.

The example below is the onCollision() method for Sayu (my main playable character). If the other object I collided with is a platform, I ask the CollisionSystem to resolve that collision for me and set some internal variables (e..g, velocity to 0). If the other object I collide with is of type "Enemy", then I deduct some health, start iFrames, etc.

```
void Sayu::onCollision(DisplayObject* other){
    //cout << "Sayu collided with type " << other->type << endl;
    if(other->type == "Platform"){
        Game::instance->collisionSystem.resolveCollision(this, other, this->x - oldX, this->y - oldY);
        _yVel = 0;
        _standing=true;
    }
    else if(other->type == "Enemy"){
        if(!this->iFrames){
            this->onEnemyCollision((Enemy*)other);
            this->curEnemy = (Enemy*)other;
        }
    }

}
```

The method above calls a sub-method called onEnemyCollision(). This is just a private method that does the actual health deduction, etc. You can see that implementation below:

```
void Sayu::onEnemyCollision(Enemy* enemy){
    this->health -= enemy->damage;
    this->knockBack(5, 10, (this->x - enemy->x) <= 0);
    this->initIFrames(120);
    this->curEnemy = enemy;
}
```

This should help get you started. For most interactions, you just need to look for collisions and have each object react to collisions appropriately. Good luck!

***Other types of interactions***

If you have other interactions that are not based on collisions, I recommend using events! Let's suppose we have a narrative text box that needs to appear once a boss is defeated. Configuring that would be pretty easy. Consider the following steps:

1. Somewhere in your code, you would have a "BigBadBoss" class (or whatever you called it). Somewhere in that class, there will be a condition that checks when the boss is defeated. When this happens, simply throw an event that let's others know the boss has been defeated. Something like:

    ```
    If(this->health == 0){
        Do stuff;
        dispatchEvent(new Event("BOSS_DEFEATED_EVENT", this));
    }
    ```

2. In your scene class, which presumably has access to your boss and the narrative text box that should appear afterwards, you can setup the text box (or some other object) to listen for the boss being defeated. Something like:

    ```
    bigBadBoss->addEventListener(narrativeTextBox, "BOSS_DEFEATED_EVENT");
    ```

3. Next, simply write a handleEvent method in your narrativeTextBox to make it visible (or tween in, or whatever) when it hears the event. Something like:

    ```
    void MyTextBox::handleEvent(Event* e){
        if(e->type == "BOSS_DEFEATED_EVENT"){
            this->visible = true;
            //position it, scale it, whatever else.
        }
    }
    ```

4. …and you are done!! With about six lines of code, we can easily hook up a text box to appear once a boss is defeated.