EECS205: Fundamentals of Computer Systems Software
Assignment 4 – Interaction
Due: Tuesday, February 28, 2017 11:59PM
Assignment 5 – Full Game
Due: Wednesday, March 8, 2017 11:59PM

Overview
This is it! We will now start to assemble a cool action video game using all of the small routines that you have been working on all quarter. This assignment is divided into two parts. The first is really a check-point (not too difficult), meant primarily to make sure that you understand what you need to do. The second part puts everything together. We will demo complete Assignment 5s on the last day of class (Friday March 10th).

Game Structure
This assignment introduces a new control structure. Understanding how this works is the first step to getting a working game up and running. Your game will consist of two new routines that our library code will call: `GameInit` and `GamePlay`. The `GameInit` routine is called by the library code once, at startup. You can use this to do any useful initializations that you may need. Once this exits, it will never be called again by the library code. The second routine `GamePlay` is the heart and soul of your game. It will be called many times per second. This is where you will process user input, draw sprites, and manage your game logic. You should probably split many of these tasks up into other helper functions, but it is up to you to decide how to organize them.

IMPORTANT NOTE: Do NOT stick a big while loop inside `GamePlay`. Your function needs to do its thing and then return so that the library code can update the screen and get user I/O.

You can use all of your routines from the previous assignments to help you draw and have objects on the screen react to each other and the user. You should also feel free to define other procedures to help you do this. For example, you might find it useful/necessary to scale a bitmap to smaller/larger sizes, flip bitmaps vertically/horizontally, or have them fade in/out. (Contact us if you are interested in these bitmap effects.)

It will likely be useful for you to define some new structures to represent your sprites. Useful things to keep in this structure might be position in the game, velocity, acceleration, rotation angle, a state for the sprite, the bitmap that should be used for the sprite. Okay, hear me out here: **You should consider using fixed point for all of the position/velocity/acceleration values in your sprites**. I know you are likely sick of fixed point, but it can be really useful here. In particular, it makes it really easy to fine tune the movement of all the objects in your game. When you need to render a sprite on the screen, you will need to convert to integer, but that's just a simple shift --- very easy.

User Input
There are two main sources of input for this assignment: keyboard and mouse. For the keyboard, we have three important global variables that you will be able to read: `KeyPress`, `KeyDown`, and `KeyUp`. You don't need to use all three of them. You can quite possibly get away with using just `KeyPress`.

`KeyPress` is DWORD which contains the virtual keyboard code for the last key which is currently being pressed. The file `keys.inc` has a list of definitions for the virtual keycodes (go ahead and include it if you like). You can use this to identify any key on a standard US keyboard. If `KeyPress` is 0, that means that no key is currently being pressed. Things are a little messy if there are simultaneous keypresses. If you hold down on one key and then press another key, `KeyPress` will contain the value of the most recent keypress. If you then release that second key, `KeyPress` will now be zero. Sorry about that. There are other ways to do keyboard input that would avoid this artifact, but they'd make a lot of other things more difficult, so we will have to learn to live with this.

`KeyDown` is a DWORD which contains the that was most recently pressed. Note that this key might have since been released. Note that depending on the behavior that you want in your game, you may not use this at all, opting instead for `KeyPress`.

`KeyUp`, is also a DWORD which contains the key that was most recently released. You might find this somewhat less useful than `KeyPress` or `KeyDown`. In fact many of you won't have much of a use for you in your games.

The mouse information is kept in structure (`MouseStatus`) which has three fields: `horiz` = horizontal position of the mouse cursor, `vert` = vertical position of the mouse, and `status` = which mouse buttons are being pressed. Take a closer look in the game include file (`game.inc`) for more details, especially for the codes that correspond to mouse button combinations.

Collisions
For this assignment, you will also write a routine that performs some simple collision detection. This is obviously useful in a video game. Collisions occur all the time, for better or for worse. We will use use a pair of DWORDs which correspond to the (x,y) coordinates at the center of sprite (screen object). We will aso use the bitmap structure (which corresponds to the current image) as a way to track the width and height of the object. When we attempt to determine if two screen objects have collided, we’re trying to see if their bounding boxes intersect.

By way of example, let’s assume that I had a screen object called **`one`** which had its center at (**`one.x`**, **`one.y`**). Let’s also say that its corresponding bitmap (named **`myBitmap`**) is pointed to by **`one.ptrBitmap`**. It would have a bounding box (an imaginary rectangle that surrounds the image on the screen) whose corners are:

Upper Left: (one.x - bitmap.width / 2, one.y - bitmap.height / 2)
Upper Right: (one.x + bitmap.width / 2, one.y - bitmap.height / 2)
Bottom Left: (one.x - bitmap.width / 2, one.y + bitmap.height / 2)
Bottom Right: (one.x + bitmap.width / 2, one.y + bitmap.height / 2)

Let’s say that I had a second screen object called **`two`** which had its center at (**`two.X`**, **`two.Y`**) and corresponding bitmap was pointed to by **`two.ptrBitmap`**. A natural thing to do would be to see if these two screen objects collide -- determine if their bounding rectangles intersect. You will therefore write:

```
CheckIntersect PROTO STDCALL oneX:DWORD, oneY:DWORD, oneBitmap:PTR
      EECS205BITMAP, twoX:DWORD, twoY:DWORD, twoBitmap:PTR EECS205BITMAP
```

Which takes object centers as a pair of (x,y) values and corresponding bitmaps. Given this information, this routine should return zero if there is NOT an intersection of these rectangles (e.g. they do not overlap) and a non-zero value in the case that they do in fact have a non-zero intersection (overlap). Think about what sorts of comparisons you will need to make to produce the right results.

Other Bitmaps

To have an interesting game, we're going to need to have some more diversity in bitmap images. We have a handy program that will take PPM bitmaps and convert them to the bitmaps we can use in this class. Yah! Also, we have a library of cool bitmaps that you can look through. We also ask you to contribute your own bitmaps.

Assignment 4 Requirements

The goal of this assignment is to serve as a checkpoint for the game and to give you some experience with two types of interaction. Your sprites will need to: (1) react to collisions with each other and (2) also respond to user input from the keyboard/mouse. These are pretty basic elements of game play. Specifically you need to satisfy ALL of the following requirements:

- Implement `CheckIntersect`
- Have at least two bitmap based sprites each of which does at least one of the following:
  - moves around the screen
  - rotates
  - animates by switching bitmaps
- Have at least one of these sprites respond to mouse
- Have at least one of these sprites respond to keyboard
- Must have some response to the collision of your sprites. Examples include, but not limited to:
  - change in direction
  - animation
  - display a message

Assignment 5 Requirements

This is the full working game. We will give you every opportunity to be creative in the theme and design of your game (e.g. you could chose a space-themed shooter, platform scroller, strategy, first person shooter). However, we have a few minimum standards that everyone must meet to receive full credit -- these are mandatory minimums. In addition, we will require you to implement at least three "advanced features". These advanced features give you an opportunity to dazzle. We will spend a little class time discussing how you might approach each of these advanced features.

<u>Mandatory Features</u>

- Must detect collisions
- Must have at least two sprites
- Game play must have reward and punishment
  - Good play rewarded with win or advance to next level
  - Bad play results in game over (or forced restart)

- Pause feature
- Respond to player input (mouse or keyboard or both)

<u>Advanced Features (Choose any three)</u>

- Special visual effect (e.g. lightning, fade in/out, static, whole screen shaking, complex explosions)
- Massive number of active, discernable sprites (100+)
- Scrolling background
- Platforms
- Sound or Music
- Gravity or Other Forces (objects fall/move with non-constant velocity)
- Multiple in-flight projectiles
- Powerups
- Complex Scoring System with either:
  - Multiple increment/decrement values to player's score
  - Buy items to improve situation (e.g. upgrades/powerups/lives/health)

Getting Started

Download and unpack the assignment files from the courseweb page. Then copy your blit.asm, trig.asm, stars.asm, and lines.asm files from the previous assignments into the assignment directory. Also copy all the related .inc files. At an absolute minimum you will only need to modify `game.asm`, but you are welcome to modify any of the files and likely will add your own file. Fill in your own code in `game.asm` to implement the two new routines. Feel free to add any other new helper routines, source files, bitmap source files, sound files, or anything else that helps you to write a cool game.

Logistics

You should hand in your assignments via Canvas. For these assignments you will likely have to make modifications to several files. You should hand in everything needed to build your assignment (makefile and all .inc and .asm files) in a single zipped folder. We will build your assignment from scratch, so make sure that the make.bat is capable of producing a single valid executable.

To receive full credit, your source file must be fully commented and must compile correctly. Your executable should not crash (e.g. due to some pointer weirdness) or cause images to wrap around the screen (nor anything else crazy).