National University of Singapore
School of Computing
CS3217: Software Engineering on Modern Application Platforms
AY2011/2012, Semester 2

Problem Set 3: **Huff & Puff Level Designer**

Issue date: 20 January 2012
Due date: **31 January 2012 23:59**

Introduction

In this problem set, you will build upon what you had done in the previous problem set to implement a simple application in order to better understand the MVC design pattern, handling gestures, and file operations/persistent storage. We also want you to consider different ways of implementing the requirements for this problem set before you decide on the final implementation.

You will also design and implement a few simple objects. Some specifications will be given, but you will not be given the full set. You are expected to add the appropriate interfaces and specifications to complete the problem set. Please ensure that you document your code according to the standards of CS3217. Each new class should have an overview and the procedures should have the appropriate clauses. Typically only the header (`.h`) files need to have the procedural comments, since people typically need only to refer to your header files to use the classes you implement. However to make grading easier for your TAs, please also replicate the comments in the `.m` files (just like you had seen in the previous problem sets).

> **Reminder:** Please read the entire problem set before starting.

There are two zip files that come with this problem set:

- `ps03-images.zip`, which contains the image files you will need; and
- `ps03-code.zip`, which contains the skeleton specifications for the game.

Please make sure you have both of them.

When programming a new game, you usually start by implementing some basic features and subsequently enhance it with more complex features. As the game grows, the code gets more complicated and the classes will get bigger. To avoid ending up with a mess of spaghetti code, we typically to decouple the rendering from the logic. Doing so will also allow you to let the game logic run at a different speed than the rendering. Actually, the game logic will know nothing about the rendering, so it does not matter how the game is displayed (2D, 3D, ASCII art or whatever). The rendering however depends on the logic, because it needs info on how/where to display everything. Typically, all the rendering is done in the *view* module.

The game logic is typically subdivided into a *model* and a *controller*. The model is keeps track of the objects in the game world, and knows nothing about displaying, user input, etc. It just implements all the world rules, and how entities interact with each other. The controller knows about the model and can manipulate it depending on the external inputs.

> **Good Advice:** Please start working on this problem set early. If not, you will live to regret.

Model-View-Controller (MVC)

In this problem set you will familiarize yourself with the **Model-View-Controller (MVC)** design pattern. In MVC, the logic of the application is separated from the user interface such that development, testing and maintenance of these modules are independent. An MVC application is a collection of model/view/controller triads, each responsible for a different UI element.

The **model** manages the behavior and the data of the application, provides information to the view about its state, and updates its stated according to the controller. The model may notify observers (such as views) when the information data changes so that they can render the updated state.

The **view** renders the model into a form suitable for interaction, typically by user interface elements. It is possible to have multiple views for a single model for different purposes.

The **controller** receives input from the user and instructs the model and view to perform actions based on the input.

The Cocoa and Cocoa Touch frameworks encourage the use of the MVC pattern. Interface Builder constructs views, and connects them to controllers via *Outlets* and *Actions*. You have already experienced a first taste of MVC through Interface Builder in Problem Set 2, where you have started constructed the view of the game you will be developing in CS3217. In the problem set, you will continue from where you stopped last time to develop a level designer for a game called Huff & Puff by adding controllers and models.

Huff and Puff Level Designer

Open the project "Game" created in the previous problem set. You should have the following:

- `GameViewController.h` and `GameViewController.m`, the *controller* for the initial view that is loaded when the game starts;
- `GameViewController.xib`, the Interface Builder file that contains the *view* elements.

In the previous problem set, you have started building the interface using two different methods: Interface Builder and programatically. Using Interface Builder, you have added the *SAVE*, *LOAD*, *START* and *RESET* buttons, and you wired them to the controller. You will notice that the top view is already wired by Interface Builder to the controller property `view`. Thus, `GameViewController` acts as a controller for the top-level view and for the game buttons. Programmatically, you have added two more views for the ground and the background images, as subviews of the `gamearea` view created using Interface Builder. Moreover, you have changed views attributes both using Interface Builder and programatically. For example, you have changed the text of the buttons and the *Vertical* and *Bounce Scroll* properties of the `gamearea` view from Interface Builder, and you have changed the *content size* property of the `gamearea` view programatically. Until now, all the connections between views and controllers have been done in Interface Builder (wiring), but these connections can be done also programatically.

The following is a brief overview of the game that you will be building called *Huff and Puff*. When launching the game, the user is presented with the interface, that is similar to the one you created in Problem Set 2, except that it has an extra palette that contains objects that the user can place in the game area. The objects that can be placed by the user includes a wolf, a pig, and different types of blocks. After dragging the game objects onto the play area and perhaps arranging and resizing them, the user can use the *SAVE* button to store the currently placed objects in a file on the iPad. The *LOAD* button can be use to load and restore the a previously saved level, and the *RESET* button will clear all placed objects from the level designer. Once the game objects are placed, the user starts the game by pressing the *START* button.

In the game, the user specifies the angle and power with which the wolf will blow a puff of air towards the pig. The air blown by the wolf will interact with the other objects in the game (blocks and the pig) according to the physics engine that you will build in the next problem set. The objective of the game is to destroy the pig that is protected by the blocks, ironically by crushing him with the blocks. In this problem set, the *START* button will be disabled. You will only have to build a rudimentary level designer.

The features of the level designer that you are expected to implement include the following:

- Reset, save and load a game level;
- An object palette from which the user can drag and drop one wolf, one pig, and any number of blocks;
- Move objects in the game area (translate);
- Rotate and resize objects in the game using touch gestures (pinch to resize, two-finger rotate);
- Delete game objects (double tap);
- Game block only: the ability to change the type of the block (single tap).

Problem 1: Create the object palette. *(10 points)*

To create the palette, you need to add a new view, `palette`, to the main view. By adding it to the main view (and not to the `gamearea` scrollview), the user should be able to scroll in the game area and the palette objects will remain in the same place on the screen. In the palette, add as subviews three palette objects, corresponding to the three types of game objects that the player can drop in the game: wolf, pig and block. You can find the necessary graphic files for the game objects in the archive for this problem set. Figure 1 is a sample screenshot of the level designer. You are however free to redesign the layout of the level designer as long as you satisfy the basic requirements.

From the object palette, the user drags and drops game objects in the game area. Please note that only one wolf and only one pig objects can be placed in the game area, but any number of blocks are allowed. The user should not be able to place more than one pig and one wolf i.e. the palette would display the pig and wolf as empty or disabled after they are dragged into the game area.

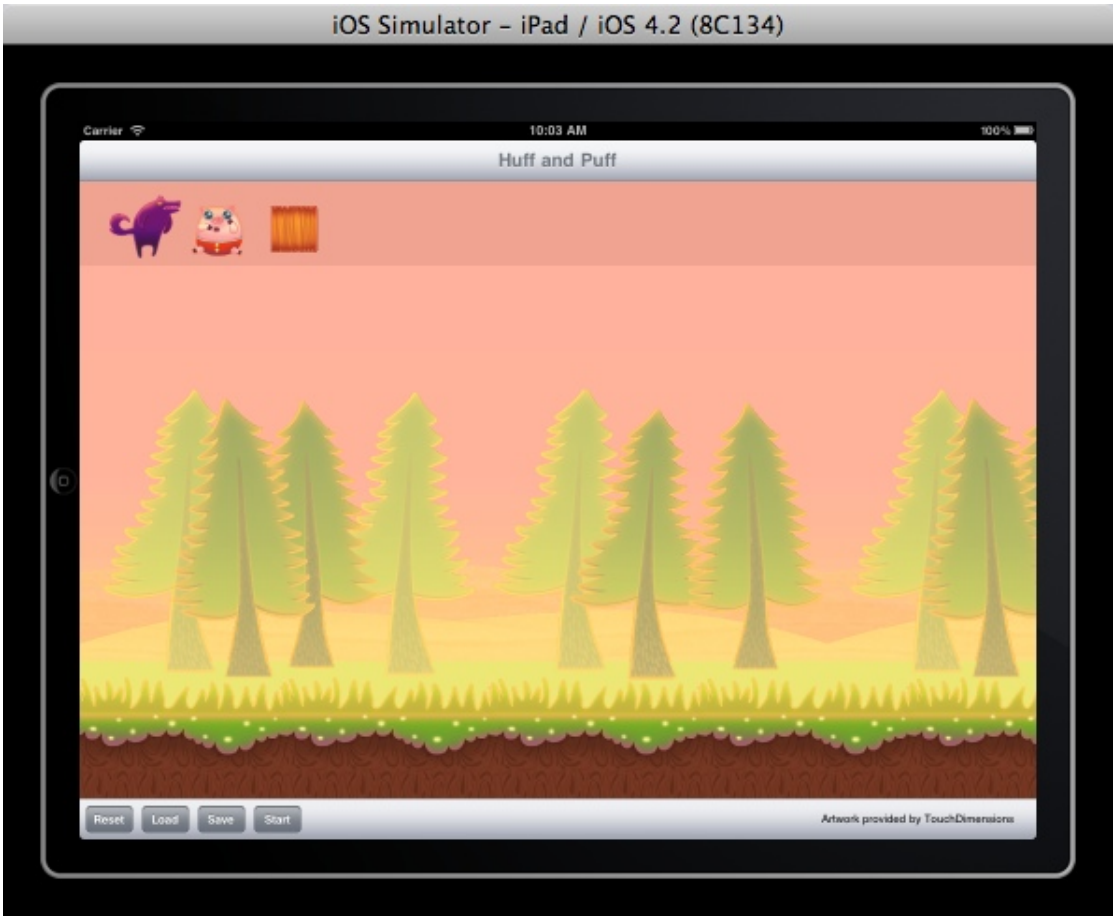

Figure 1: Game loads with the designer view. The palette contains game objects that can be dragged into the game area below.

Problem 2: Explain Your Design. (40 points)

> More thinking, Less Coding.
> *- Ben Leong*

Before you begin, please spend some time understanding the requirements of this problem set. Think about the following questions: What objects do you need to implement? How will the objects interact with the `GameViewController`? How do you best organize the code for these objects? Are there alternatives? How will you add new game objects?

Once you have decided on the best way to organize your code, please answer the following questions by including a PDF file `design.pdf` at the root directory of your project folder.

a. How did you apply the MVC pattern in this problem set? Explain how you decided to organize the code for your game objects with an entity-relationship diagram for your implementation of the game. With the entity-relationship model diagram as a start, build a module dependency (class) diagram for the entire project. The entity-relationship model is an abstract and conceptual representation of your data. An entity is a unique thing that can exist independently in your design. Each entity must have a set of unique attributes. A relationship captures how two ore more entities are related to each-other. Relationships also may have attributes. The entity-relationship diagram shows graphically the entity and the relationship sets, including the

cardinality of the relationship (e.g. one to one, one to many, many to one). Please follow the conventions as discussed in lecture. *(10 points)*

b. Explain why you decided on your chosen design over alternative designs. *(5 points)*

c. How would you add a new "projectile" object - the wolf's breath - to the game? (To be done later) *(5 points)*

d. How do you plan to integrate the game objects with a physics engine? *(10 points)*

e. What other functionalities do you think you might want to add to the game later and how can you extend your code to support these new features, other than what has been specified in this problem set? Give two concrete examples (changes to your code as well as the structure of your classes). *(10 points)*

Self-check: A correct implementation of the MVC framework has the following properties:

- the model knows about nobody
- the view knows about the model (but accesses it only through the controller)
- the controller knows about the the view and the model
- the controller observes the view

Problem 3: Implementing the Game Objects. (90 points)

You will be using the MVC pattern to implement the game objects. Thus, a game object is represented by the triad: model, to store the state of the object; view, to represent the object on the screen; and controller, to manage the model and the view.

- **Model**. There are two important aspects regarding the object model. In this problem set, you are concerned mostly with the position of the object on the screen, such that you can save and restore the object to that position using the buttons created in the previous problem set. However, be aware that for the next problem set the game objects state will be more complex as you develop a representation required for the physics engine. *(10 points)*
- **View**. Rendering to the screen is handled by the view. It uses the model to know where to draw everything. The view doesnt have any other functionality than this. You are provided with the required sprites in the attached archive. A *sprite* is an image or animation that is going to be integrated into a larger scene, such as our game. Thus, you will find included both image sprites, and animation sprites. *(20 points)*
  - `wolfs.png` is an animation sprite. It is a 1125x449 pixels image that contains 15 animation frames. The animation is that of the wolf sucking air (frames 2-9) and then blowing

air (frames 10-15) and will be used only during the game. The first frame you will use for the (default) image representation of the wolf. The frames are arranged in 5 rows and 3 columns. The default size of the wolf image frame is thus 225x150 pixels.

- `pig.png` is an image sprite. It is a 55x55 pixels image. For the game, use a default view size for the pig of 88x88 pixels.
- `straw.png, wood.png, iron.png, stone.png` are image sprites. These are the images that you will use for the game blocks. The first image represents a *straw* block, the second image is for a *wood* block, the third for a *iron* block and the last for a *stone* block. The images come in two sizes with two aspect ratios: 50x50 pixels, and 15x65 pixels. For now, we will be using only one type of block with aspect-ratio of 15:65. Thus, the square blocks will be scaled to that aspect ratio. The default size of the block frame is 30x130 pixels. By tapping on the block, the type of the block is changed in a round-robin manner between straw, wood, iron and stone.

> **Note:** If you use the images in their default sizes, the palette will have to be very big. Thus, the views placed in the palette should be scaled down to a reasonable smaller dimension, and when the user drags the object from the palette to the game area, the object should be scaled back to the default size specified above, and animated. To make it easier for the user to place the blocks, you can use a square shape for the palette, which is then scaled to its original aspect after the user drags the block in the game (see the screen-shot from Figure 2). This will introduce you to the basics of animation that you will be required to do in the next two problem sets.

- **Controller**. The controller handles the user input and manipulates the model. First it checks for user input, then it might query the view to see which on-screen objects are being manipulated by the user, and finally it changes the model accordingly. Since all objects respond in a similar way to the user input (for example: drag from palette, resize using pinch, rotate, translate in the game area), you are given the interface for an abstract class representing a game object controller. You need to complete the implementation of the abstract class (*30 points*) and then create subclasses with the additional requirements for each object type (*30 points*). The interface is found in the file `GameObject.h`. The subclasses implementation should contained in the files `GameWolf.m`, `GamePig.m` and `GameBlock.m` respectively. *(60 points)*

Touch Gestures. To implement the translation, rotation, and zoom, you can use *gesture recognizers*. For translation, you can use `UIPanRecognizer`, for touch you can use `UITapGestureRecognizer`, for zoom you can use `UIPinchGestureRecognizer`, and for rotation you can use `UIRotationGestureRecognizer`. An example use of gesture recognizers is the following:

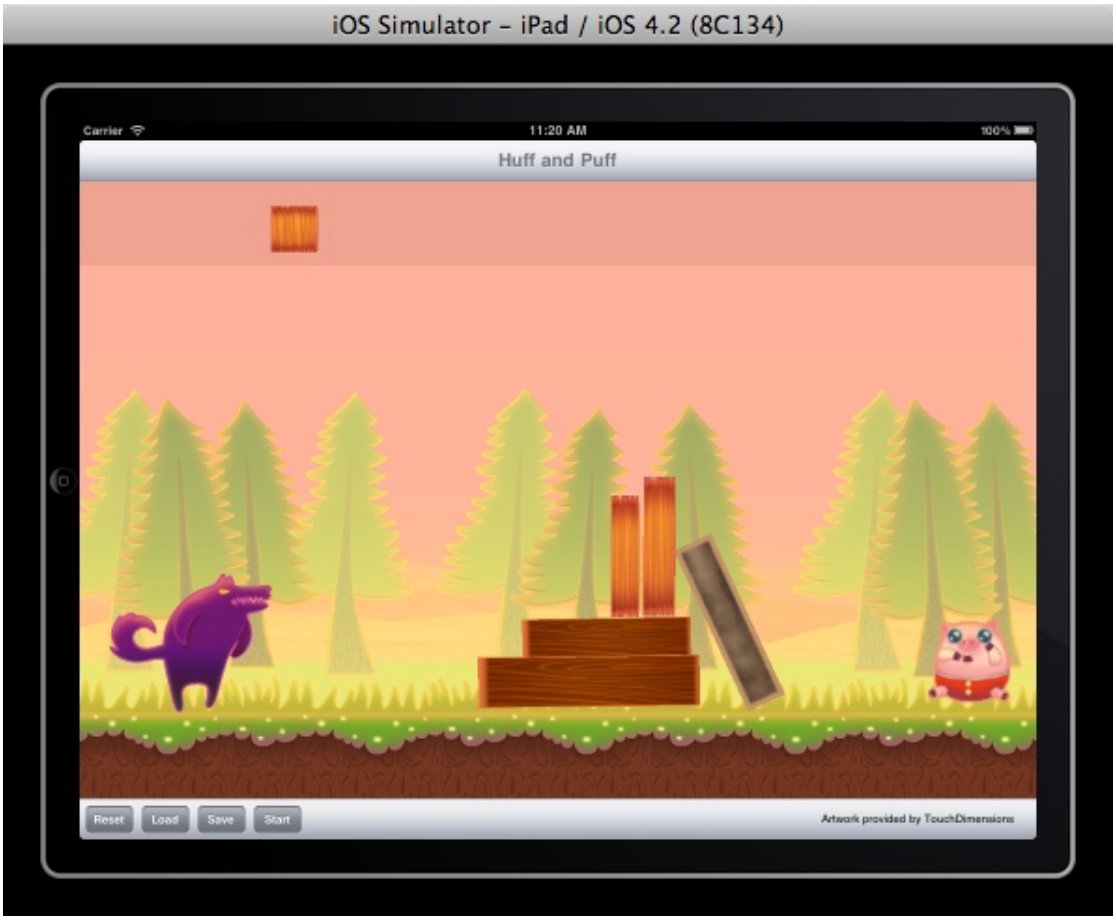

Figure 2: Example level created in the designer. After placing the wolf and the pig, they are removed from the palette, which now contains only block objects. Several blocks are placed, which are rotated and scaled. The block object has aspect ratio 1:1 in the palette and 15:65 in the game area.

```
// This is required for touch interaction with the view
view.userInteractionEnabled = YES;

UIRotationGestureRecognizer *rotation = [[UIRotationGestureRecognizer alloc]
initWithTarget:self action:@selector(rotate:)];
[view addGestureRecognizer:rotation];
```

Supported Operations. Each of the game objects needs to implement the following operations:

- Drag and drop from object palette to main game area;
- Translation (drag);
- Rotation and resizing (pinch to resize, two-finger rotate);
- Deletion and returning to the object palette (double tap);
- Game block only: the ability to change the type of the block (single tap).

Problem 4: Saving and Loading Game Levels. *(50 points)*

Finally, you have to implement the reset/save/load functionality for the level designer. You are to design your format for storing the game objects and decide how you want to store the objects. You can implement the required behavior by wiring the `buttonPressed:` outlet to the interface defined in the class extension `GameViewControllerExtension.h`. Make sure you implement the ability to save and load from different files, as well to modify and re-save a level. Please explain how you chose to implement the save/load function in `design.pdf`. Please argue that your implementation is the best one among all the alternatives you considered.

There are five ways to maintain data persistency on iOS:

- Property lists
- Object archives
- Manual encoding
- SQlite database
- Core Data

In this problem set, we will briefly discuss the first three. You should go and learn about the other two ways on your own.

All applications in iOS are sandboxed. As such, applications can only read and write data from several folders. The `Documents` folder should be used for persistent data since it will get backed up regularly. The `Cache` folder can be used to store information available to the application between launches, but it's not backed up by iTunes. Lastly, the `tmp` folder is available only when the application is running. You can find the path of the `Documents` using the `NSSearchPathForDirectoriesInDomains()` function and append that path to your file names.

You can choose to save your data either using a single file, or using multiple files. We will discuss only the case of single file since it might be easier to implement. You start by having one root object (e.g. an `NSArray` or `NSDictionary`) which you populate with all the data that have to be persisted. When saving, you rewrite the single file with the contents of the root directory.

Property Lists. Property lists are convenient because their representation is in XML. As such, you can view and edit files manually using *Xcode* or the *Property List Editor* application. You can create and write both `NSArray` and `NSDictionary` containing any object as long as the objects are serializable. Although any object can be made serializable, only certain objects can be placed in the Objective-C collection classes, such as: `NSArray`, `NSMutableArray`, `NSDictionary`, `NSMutableDictionar` `NSData`, `NSMutableData`, `NSString`, `NSMutableString`, `NSNumber`, `NSDate`. If you can build your model using only these objects (NOTE: look at the documentation for `NSData`) then you can save your entire game data using the collection method `writeToFile:atomically:`.

Object Archives. One of the problems with property lists is that custom objects cannot be serialized. In Cocoa, "archiving" refers to a more generic serialization that any object can implement. In fact, any object model should support archiving since this allows you any model object to be saved and restored. As long as every property you implement in your class is either a scalar (`int`, `float` etc.) or an object that conforms to the `NSCoding` protocol, you can archive your objects completely. To make an object conform to the `NSCoding` protocol, you must implement two methods, one to encode your object into an archive, and one to create your object by decoding it from a file:

```
-(void)encodeWithCoder:(NSCoder*)coder {
    // This tells the archiver how to encode the object
    [coder encodeObject:self.strVar forKey:@"theStringVariable"];
    [coder encodeInt:self.intVar forKey:@"theIntVariable"];
}

-(void)initWithCoder:(NSCoder*)decoder {
    // This tells the unarchiver how to decode the object
    self.strVar = [decoder decodeObjectForKey:@"theStringVariable"];
    self.intVar = [decoder decodeIntForKey:@"theIntVariable"];
}
```

To actually store your data then your have to create an instance of `NSMutableData` to hold the encoded data and then create an `NSKeyedArchiver` instance to save the data to a file:

```
NSMutableData *data = [[NSMutableData alloc] init];
NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
initForWritingWithMutableData:data];
[archiver encodeObject:model forKey:@"modelKeyString"];
[archiver finishEncoding];
BOOL success = [data writeToFile:@"/path/file" atomically:YES];
```

Restoring an object from a file is similar, using an `NSKeyedUnarchiver` object and its `decodeObjectForKey:` method.

Saving images: Because the `UIImage` class does not implement the NSCoding protocol by default, you need to extend it using an Objective-C concept known as *categories*, and add these methods yourself. As before, to encode an object you would do:

```
NSData *imageData = UIImagePNGRepresentation(self);
[coder encodeObject:imageData forKey:@"image"];
```

And to decode it:

```
NSData *imageData = [coder decodeObjectForKey:@"image"];
return [[UIImage imageWithData:imageData] retain];
```

Manual Encoding. The general idea in this approach is to open a file and to pass the handler to each object for the object to write itself into the file using a custom format. You will need to assign a unique identifier (type) to each object and write the type of the oject to file before calling the object to write itself. To reconstruct objects, you will require a factory function that will read the file for the object type, create an associated empty object and then pass the file handle to the object for the object to initialize its state. Effectively, each object must support a `read` and a `write` method (which you will have to define yourself).

Problem 5: Testing *(30 points)*

Testing is an integral part of software engineering. Since you are not implementing any complicated ADTs in this assignments, unit tests are probably not particularly helpful. That said, you are supposed to implement a large number of features and it is important for you to test your final code to make sure that your application meets the stated requirements. The way to do this is to start from a hierarchy and then break down into smaller and more specific cases. For example:

- Black-box testing
  - Test implementation of file operations
  - Save
  - ...
  - Test implementation of game objects:
    - Drag from Palette
      - Wolf
      - Pig
      - block
      - ...
    - ...
- Glass-box testing
  - ...

Please come up with you testing strategy describe it in `design.pdf`. Of course, you should actually test your application as you have described instead of justing listing down what you think you ought to test! :-)

Bonus Problem: Reflection (3 Bonus Points).

Please answer the following questions:

a. How many hours did you spend on each problem of this problem set?

b. In retrospect, what could you have done better to reduce the time you spent solving this problem set?

c. What could the CS3217 teaching staff have done better to improve your learning experience in this problem set?

Your answers to these questions should be appended at the end of `design.pdf`.

Grading Scheme

In this module, you are training to become a good software engineer. The first and basic requirement is that your code must satisfy the requirements and be correct. Above and beyond correctness, you are required to write well-documented code. In real software projects, just ensuring that your code can do the job is **not** sufficient. Remember that if you are doing anything useful at all, code has to be maintained and the probability that some poor soul will have to come along to read and modify your code is very high. Your goal is to minimize the grief of this poor fella and make him love reading your code.

For this problem set, testing/grading will be done by running the project in the simulator. You should also probably upload the application to your iPad to make sure that the gesture code works correctly. We will be looking at the following:

- Your submission should adhere to the submission format.
- Your project should build without errors or warnings.
- Your project should run without crashing.
- Each of the sections above will be considered to verify that youve completed the problem set correctly.
- You have answered all of the above questions.
- Your code should be well-documented, correctly indented and neat. You should not use magic numbers.

Points will be taken off if you fail to comply with these requirements.

Mode of Submission

Like the previous problem sets, the teaching staff will be grading your code directly on GitHub. Your solution for this Problem Set should be contained in a single directory called `ps03`, which should be inside the root directory of the private repository assigned to you. The `ps03` directory should contain **all** your project files. However, in order to keep the size of your submission small, you may omit the `build` subdirectory that contains the compiled binaries. In addition, your submission directory should contain your design explanation `design.pdf`, and a README file where you specify which implementation files contain your code.

You must upload all your work to the master branch of the remote repository. You will be graded on the **latest commit** before the deadline.

Clarifications and questions related to this problem set may be directed to the IVLE Forum under the heading “Problem Set 3: Huff & Puff Level Designer”.

Good luck and have fun!