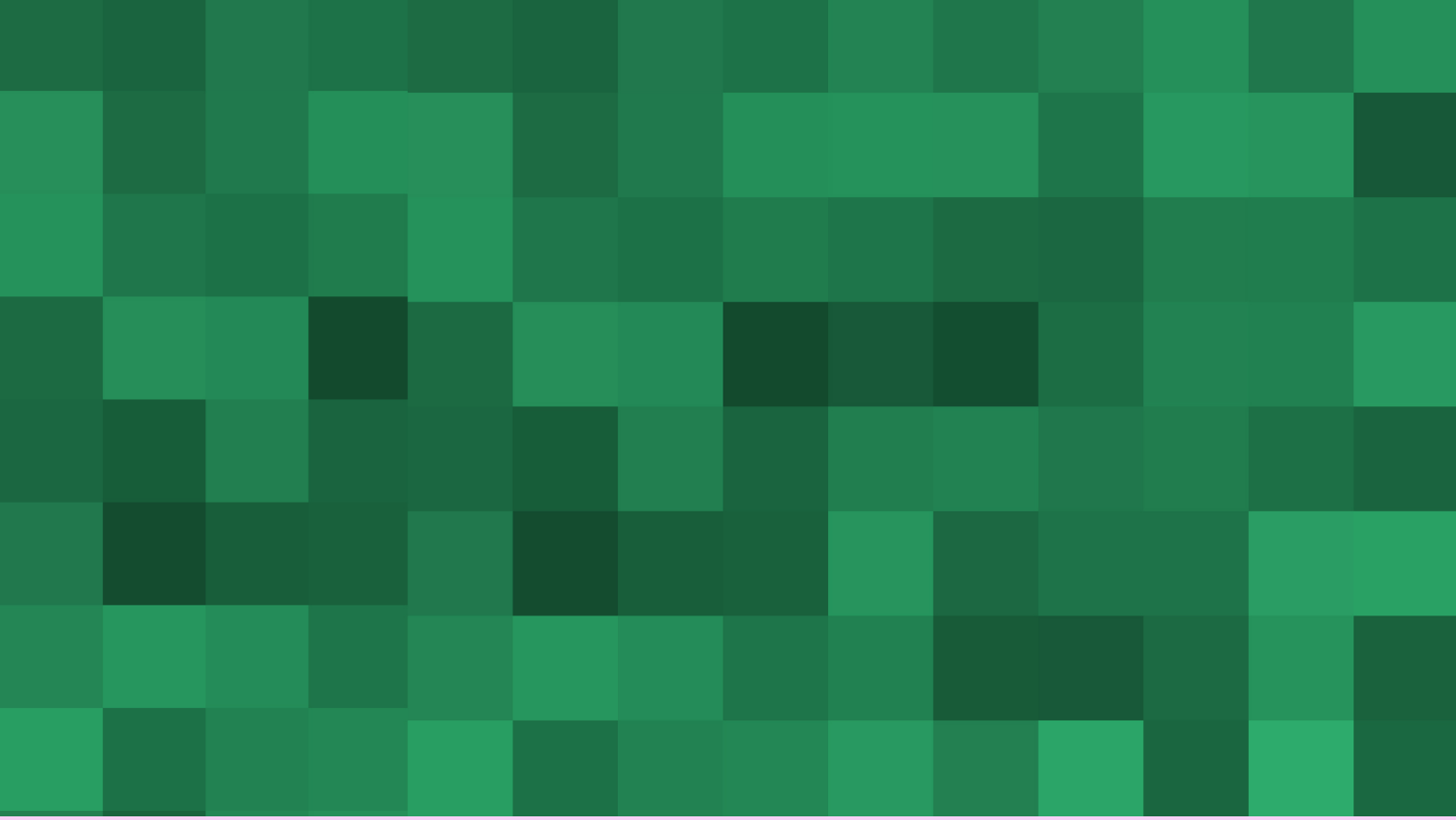
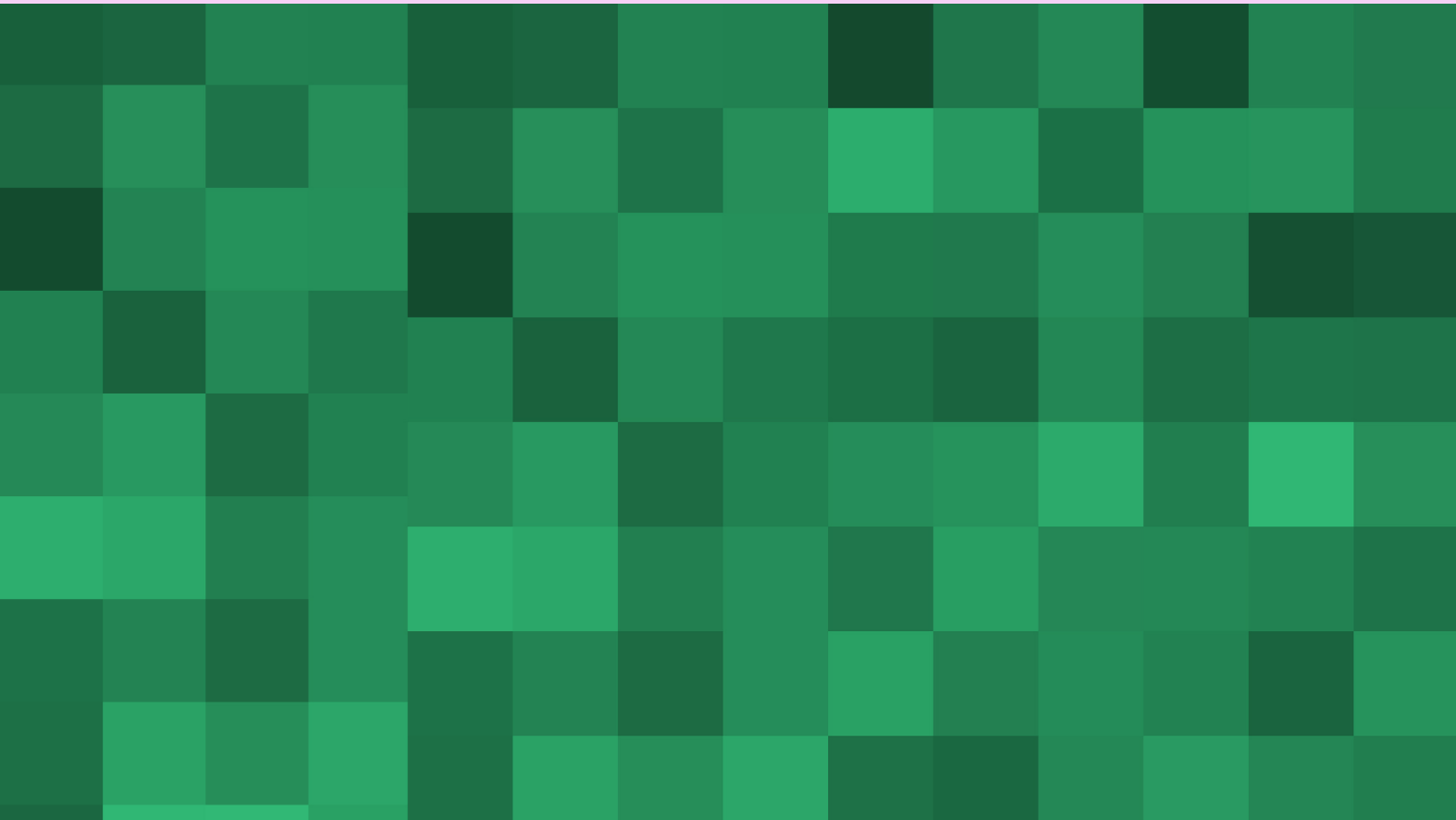


CS-224 Objected Oriented Programming

Project Guide

Habib University



PUBLISHED BY HABIB UNIVERSITY

Written by Ahmed Ali and Aiman Khan

First issue, July 2018

Contents

I Getting Started

1. LazyFoo Tutorials

Before we start on working on this project, you should check out these tutorials by LazyFoo . LazyFoo provides a guide for SDL which will be used extensively throughout this project. It is recommended that before you proceed in this guide or begin your project, have a look at those tutorials. LazyFoo tutorials are text based and provide sample code for each tutorial. It is an excellent guide that starts from a step-by-step guide for setting up SDL for your preferred IDE and covers everything you will be needing for this project. You may also learn SDL from another source you would prefer, for which a list of all tutorials you should cover at the minimum are provided below :

1. Hello SDL
2. Getting an Image on the Screen
3. Event driven programming
4. Key presses
5. Extension Libraries and Loading Other Image Formats
6. Texture Loading and Rendering
7. Clip Rendering and Sprite Sheets
8. Animated Sprites and Vsync
9. Rotation and flipping
10. Mouse events
11. Key states
12. Sound effects and music
13. Motion
14. Scrolling
15. Collision detection

2. Constructing a UML for your project

2.1 Purpose of UML

It is impossible to work efficiently in a project which has many different aspects and people working on it to be done efficiently without everyone having a complete understanding of what to do. UML stands for unified modeling language, it is an important tool for representing different parts of your project. Understanding and being able to create one will be an important skill for everyone in a project and therefore you will be creating one for your project as well. A good and detailed UML is as important as a blueprint is for a building, an architect needs to be able to clearly convey to the engineers his design and ideas otherwise the building may turn out to be much different than what it was perceived to be. For this task, we are particularly interested in creating a structural diagram for your project, specifically, the class diagram. You can learn more about creating a class diagram UML from the links provided in the recourses and tools section.

2.2 Deciding classes and attributes

An important reason for asking for a UML right away is so that you may decide before starting work on your project what it is that you wanna create. In this UML you will have to decide on all classes you think your project will have as well as their attributes. Brainstorm with your team what would be the game mechanics and how you would be implementing them. You need to decide upon the classes for your games, their relations, even their functions, which need to be public and which need to be private. Your experiences in labs should have equipped you to understand what should be included in the player class for example. Think about all your classes and what variables will you need in them to create this UML. Give this UML a lot of thought, the better the UML, the easier it will be to code all the classes in your project. Keep in mind, you will have to also submit a final UML for your project when you submit it and may be judged on how close the two UMLs are, the closer they are, the more it shows that you had a clear idea of what you wanted to do.

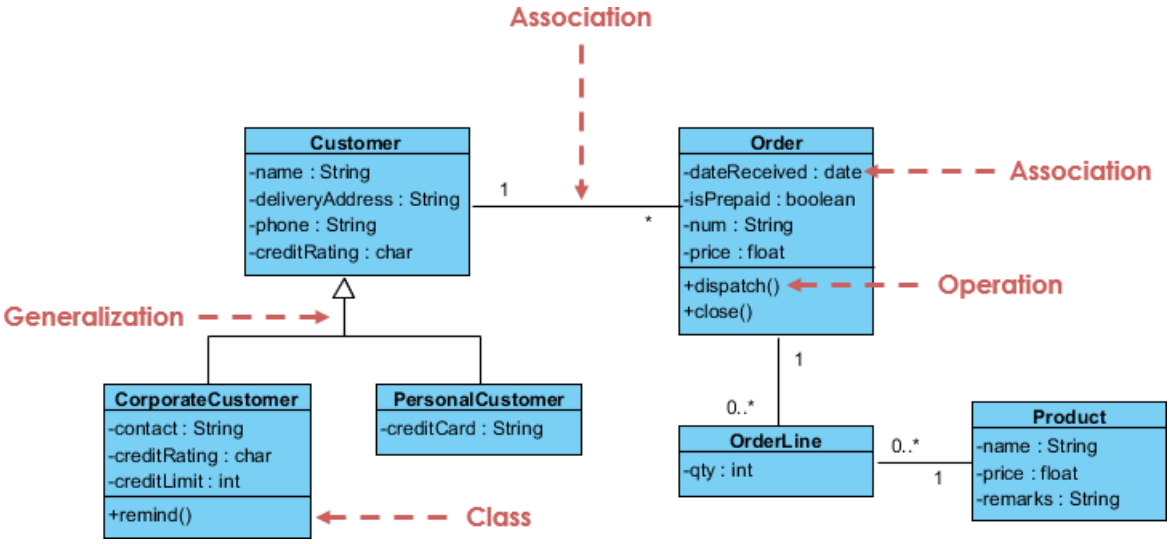


Figure 2.1: Sample UML class diagram

2.3 Class relations

From all your practices about class relations, there should be no more need of explaining the benefits of using inheritance in your project. Clever use of inheritance in your classes will save you from rewriting a lot of code from your projects as well as give you more flexibility on incorporating many kinds of objects, in a single data structure for example. Other kinds of class relations are **Association, Aggregation, Composition and Dependency**. How well you define these relationships and appropriate use of relationships depending on your use will also be an important criteria in judging your project.

2.4 Dividing work

Along with defining classes and relationships for your project, you will also be asked to show how this project will be divided among all team members. The UML can easily represent the work you divide and it is recommended that this information be conveyed in your UML through different colors, which improves readability. You may color each class by a color which represents a group member. It is important that at this stage, you have a clear idea about how much work each class is, as well as the weaknesses and strengths of your team so you can divide work accordingly. Similar classes should be awarded to the same person so that work can speed up.

2.5 Resources and tools

- What is a class diagram? For a deeper understanding and tutorial for making your UML
- UML creation tools:
  1. UMLet
  2. Gliffy
  3. https://creately.com/Draw-UML-and-Class-Diagrams-Online
  4. www.lucidchart.com/

II Visuals and Audio

3. Designing Sprites

3.1 Choosing sizes

3.1.1 Game Screen

Scaling a game through multiple displays is always a tricky job. A certain size may be fitting the entire window in a 720p display but appears to be too small on a 1080p display. Simply increasing the size of your window is not a viable solution as that would cause it to overflow from the screen on smaller displays. It is therefore a good idea to fix your screen size to 1024 by 768 pixels which is a gaming standard and it is a much greater chance your window fits on the screen of your computer. If it is too small for your or your partner's display, you should reduce your screen resolution which would make the window appear larger on a higher resolution monitor. Figure 3.1 below will give you an idea about how different resolutions' sizes vary relative to each other.

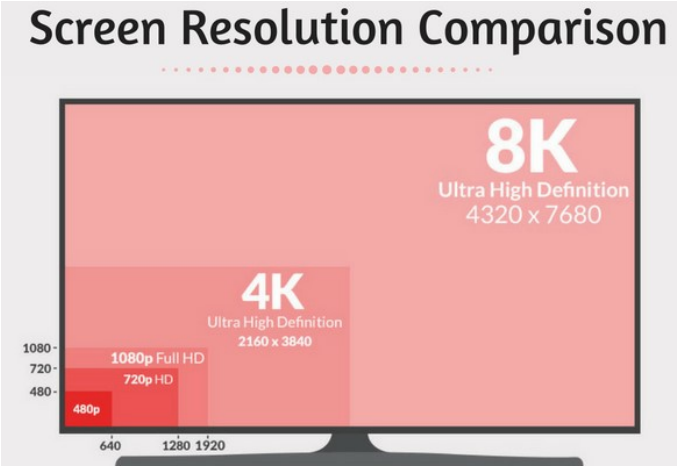


Figure 3.1: Resolution comparisons

3.1.2 Sprites

To save memory and use it as efficiently as possible, it is a good idea to keep your graphics in a size that is a power of two. The size could vary for every object and game. It is important that the sprites have a transparent background without which no game can look good, if the backgrounds of the sprites are not transparent, you will be only available to render rectangles on the screen. For moving objects it you will have to make multiple sprites for them depending on how do they move, in general, the more frames for animation you create, the smoother your animation is.
While you may choose to not do so, all professional developers always design and produce their sprites in a much higher resolution than the resolution they need. For example, a sprite with dimensions 32*32 may be made originally in 512*512 size and then shrunk down. This results in a much higher quality image when shrunk to it's required size as the details it is designed in can be much greater. It also gives the advantage of being able to increase the size of the sprite later on as it is still being shrunk down from original instead of being expanded from a smaller size.

3.2 Format

You should always store your images in the png file format as unlike jpeg, it allows backgrounds to be transparent, opaque or translucent.No background means that the inside the size that your image is, which is a rectangle, anywhere there is no image,for example, the region outside a circle that extra area is transparent and therefore it is whatever is below it.Without opaque backgrounds, your object can be of any shape, not just a rectangle which is a necessity for your project. There are a number of tools that you can use to remove a background and some of them are mentioned below, in the recourses and tools section. Figure 3.2 shows the effect of a transparent background.

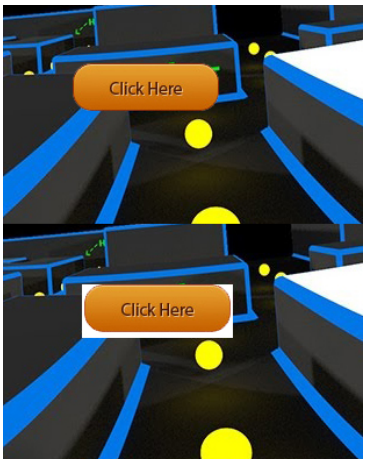


Figure 3.2: White vs transparent background

You may realize this that it is never the case in any game or software that individual objects are rendered without a transparent background. If you are making your own graphics or downloading them from the internet, you should remove any background included in the image. One important point to remember here is that if an image has a transparent background (which is likely the case if you download sprites from the internet with a .png extension) opening and saving in MS Paint will make that background white, so never edit such images in paint, unless you can remove its background later.

3.3 Making your own vs Getting from the internet

If what you are trying to make has been done before and is popular, there is a good chance someone has uploaded it's recourses on the web. Unless you are skilled in it, making sprites from scratch is a very difficult job to do, one that may take dozens of hours depending on how detailed and many sprites you have. It is therefore a good idea to always search for sprites you want on the internet first. That said, even if you do find sprites you need, they may be not exactly what you want or more commonly, of very poor quality. You may need to edit some or even make new by yourself. Even if sprites may not be readily available, you may still easily find back drops for your project in very high quality on the internet so that is one thing you will not have to make on your own at the least. If you do find sprites that aren't as good as you wanted them to be, editing over them is still a better way to do it than completely making them on your own if time is your primary concern. The spritesheet for this project was made on Piskel which is an online tool, and there are many similar websites that help producing sprites and even show resultant animation of the sprite as well as have available, many sprites as templates. Before you start on your sprites, it is recommended that you atleast spend some time on the internet searching for your desired sprites.

3.4 Sprite Sheet

Since your project will have many sprites, typically in the dozens, it is a much better idea to put all of your project sprites into a single image, which we will be calling the sprite sheet. There are many advantages of doing so:

- Sprite sheets load quicker since we do not need to load a new file every time we render object.
- A single image file will use less space since there will be only one header rather than dozens (there is alot of data stored in an image file other than just the pixels, this data is called a header).
- Individual sprites can be easily taken out so it does not complicate code either.
- The risk of putting in incorrect file names is drastically reduced.
- If there is an encoding error or you need to resize all your sprites, it becomes a much easier task to do.

Therefore, it is required by you to abide by the same procedures and maintain a single image file for all your sprites for this project. Your backgrounds and other similarly large images should be in a separate file however, since they are only being used once and then cleared from memory. All characters,numbers and buttons will also be included here.

Figure 3.3: Sample Sprite sheet

3.5 Buttons

In GUI, buttons are a very important form of user input. Our screens are flooded with buttons and you have probably never come across any application that doesn't contain buttons. For this reason, you shall also be required to use buttons in your project, especially on your home and your pause screen. Text will also be rendered on a button, not through a library but you will be rendering individual characters on each button. Your buttons should have three sprites associated with them,each representing three states. The first state is the default state which will be rendered when there is no mouse activity on the button, the second state is hover, which is active when the user's mouse is inside the button and the third sprite is rendered when the button is clicked. We will go through how to use the button later on in this guide. Buttons with all these three states can be easily found on the internet however, they will be more commonly available as a photoshop document, a simple search on Google "buttons.psd" should take you to numerous websites that have these available for free.

Figure 3.4: Sample Button sprite

3.6 Backgrounds

It is also required that you gather a plurality of background images for your project which fit your entire screen. You can easily find relevant images for your project on the internet. You will need to provide background graphics for your splash screen, your home screen, pause screen and exit screen, more on these screens later. You may choose to use the same images for all these screens although it is not recommended, more images aren't difficult to find nor add but they do look much nicer and add to the appeal of your project.

3.7 Resources and tools

- Inkspace A picture editing tool
- Photoshop A picture editing tool
- Photoshop CS6 by thenewboston for a beginner's tutorial for photoshop
- https://www.freepik.com/ contans many graphics in the psd format for easy editing and use.
- Piskel A free and simple online sprite creator
- Spriter A more advanced sprite maker

4. Using Sprites with SDL

4.1 LTexture class by LazyFoo

SDL offers a great texture rendering API which enables us to load texture and then render them to screen. There are SDL functions available that you can use to load the image and render them to screen at any position.

If you are learning SDL from LazyFoo tutorials, then you must have noticed that to make life easier, they have defined a separate class *LTexture* for image loading and rendering purposes and doing all sort of stuff that you want to do with the image like color keying etc. Making a class would make it easier to do all these things rather than using SDL functions every time.

This class has a constructor/destructor pair, a loader function that loads the image from a file, a deallocator that frees the image, a renderer that takes in a position, and functions to get the texture's dimensions.

If you are starting to work on your project, then copy paste *LTexture* class from LazyFoo tutorials, define separate header and `.cpp` for it and then continue loading images and rendering on screen.

4.2 Loading sprites

By now you have probably some idea that we define and load texture in `loadMedia()` function in `main.cpp`. If you are working with *LTexture* class, then you need to define LTexture's class object in main, and then call its image loader function `LoadFromFile(string filename)` in `loadMedia()` function.

```
LTexture gSpriteSheet; //Ltexture's object

bool loadMedia()
{
  //Loading success flag
  bool success = true;
  if(!gSpriteSheet.LoadFromFile("Images/mySpriteSheet.png",gRenderer))
  {
    printf( "Failed to load sprite sheet texture!\n" );
    success = false;
  }
    return success;
}
```

Listing 4.1: Using LTexture to load images

4.3 Rendering Sprites

This was just about loading an image. As discussed above, it is a good practice to put all of your images in a single sprite sheet to save memory. For this we will always load a single sprite sheet which will have images for all, for example, pacman, ghosts, candies and lollipops etc.

In order to render any object, for example if you want to render a lollipop to screen, you will have to define an `SDL_Rect spriteClip` as an attribute in *Lollipop's* class that will define the `x, y` positions and width and height of the part of the sprite sheet that you want to give to the object Lollipop.

The `x, y, width, height` dimensions of the clip can be figured out using MS paint. In the *View* tab, check on *gridlines* so that gridlines can appear on your image as shown in the image below. Now you can easily figure out the clip dimensions by hovering the cursor over the clip.

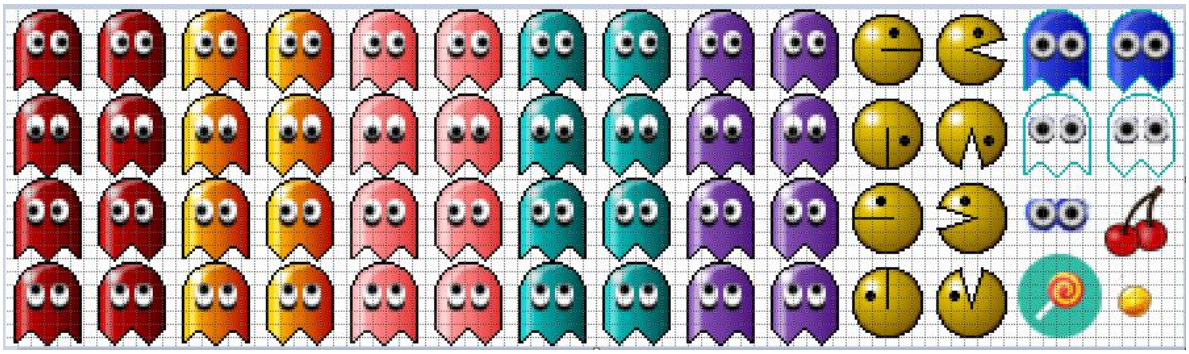


Figure 4.1: Using MS paint to figure out clip dimensions

Now that you know the clip dimensions, you can specify the values for `SDL_Rect spriteClip` in *Lollipop's* constructor which will have the `Ltexture*` as a parameter so that it knows what image needs to be clipped.

```
Lollipop::Lollipop(LTexture* image,int x,int y):FoodItem(image,x,y)
{
    //ctor
    spriteClips.x = 385;
    spriteClips.y = 93;
    spriteClips.w = 32;
    spriteClips.h = 32;

    widthSheet = spriteClips.w;
    heightSheet = spriteClips.h;
    name = "LOLLIPOP";
}
```

Listing 4.2: Setting Sprite's dimensions;

In order to render Lollipop, call LTexture's render in Lollipop's render funtion:

```
void Lollipop::Render(long int& frame, SDL_Renderer* gRenderer)
{
    spriteSheetTexture->RenderTexture(position.x-widthSheet/2,
    position.y-heightSheet/2,gRenderer,&spriteClips, SDL_FLIP_NONE);
}
```

Listing 4.3: Setting Sprite's dimensions;

5. Audio

5.1 Audio in a game

Audio is a very important aspect of a game. Audio together with graphics is what makes up the aesthetics of a game.

If you want to set up SDL Mixer, follow the instructions given in this link: Setting up SDL Mixer
If you are confused how to play sounds effects and music using SDL, then check out this link: Playing Sounds and music using SDL

5.2 Resources and tools

Following links may be useful to find free sound effects and music for your game:

- freesound

II Player(s) and other objects

6. User input

Without any form of input, your project will be basically nothing more than a video. User input needs to be handled very well and needs to be as easy for the user as possible. Your game can have any form of user input available. Most commonly it will be a combination of a keyboard and a mouse but you may choose to also include options for a joystick or even a touchscreen(necessary if you wish to develop for a mobile platform). For the sake of this guide, we will only focus on keyboard and mouse inputs. The tutorial on using mouse and keyboard is already present in the lazyfoo tutorials, which you were expected to already go through therefore this guide will skip ahead to some explanations and common traps for inputs and ways to avoid them.

6.1 Mouse

There are only two things you need to check when dealing with a mouse, it's state and it's position. All mouse events will be handled by SDL_Event.

```
SDL_Event e;
while( SDL_PollEvent( &e ) != 0 )
{
    if( e.type == SDL_MOUSEMOTION || e.type == SDL_MOUSEBUTTONDOWN
        || e.type == SDL_MOUSEBUTTONUP )
        {
            //Get mouse position
            int x, y;
            SDL_GetMouseState( &x, &y );
        }
}
```

Listing 6.1: Getting mouse state

The while loop will keep running until all registered poll events have been checked. It ensures

that no event gets skipped during processing of the previous one. This code will give you the position of the mouse anytime it is moved or a button pressed or released.

It is recommended that this be done in the main function. Lazyfoo passes a pointer to the SDL_Event to the object, which doesn't make a difference if you have only one object but if you have many, objects, this strategy makes you redundantly check the mouse state. Therefore it is better you do this once in main and then either pass the mouse co-ordinates to each object to check if the activity is over that object or get the position of each object and check that in main.

Either way, it would be incorrect to simply hard code positions to check for a mouse collision, below is an incorrect implementation :

```
if(x>=70 && x<=470 && y>=137 && y<=217){
  if(event.type == SDL_MOUSEBUTTONDOWN){
   //your code here
   }
 }
```

Listing 6.2: Checking incorrectly if a button is pressed

This code checks if the mouse is clicked inside the dimensions of a button but the problem here is that this will have to be done for each object, moreover if the position or the dimensions of the button is changed, this piece of code will have to change as well, which becomes tedious and elements the purpose of using objects entirely, below is a sample code of how to do this correctly :

```
if(button.isOverMouse(x,y)
{
    if(event.type == SDL_MOUSEBUTTONDOWN)
    {
      //your code here
    }
}
```

Listing 6.3: Checking correctly if a button is pressed

In this example button is an object for a Button and it has a function `isOverMouse(x,y)` that returns a **bool** value depending on whether the mouse co-ordinates are inside the limits of the button.
The order of your *if conditions* may be different depending on how you prefer or need.

6.2 Keyboard

When discussing keyboard inputs, there are two different methods that can be used. The first type is to use SDL_Event and figure out which key was pressed. The other type is by using currentKeyStates which uses an array of all inputs to show which one is currently pressed. LazyFoo has tutorials for both of these methods.The first method uses a queue to store all key presses and therefore using this method ensures that all the keys pressed are registered and not missed. This method may be helpful in an application which requires something to be typed. However, this is not useful in making games and will most likely cause problems as a key pressed may register any number of unexpected additions to the queue and may cause delays in input and processing. currentKeyStates however show the keys pressed in real time so a key pressed is registered as long

as it is pressed. This method also comes in much handy when checking for multiple key presses at the same time, which may be the likely case if you are building a game that has two players. Whichever method you choose, it is recommended that this checking for key presses be in your game's main loop.

7. Player

7.1 Assigning control to the player

One of the most important aspects of a game is the control assigned to a player. It may entirely depend on the type of the game whether the player should be given a complete and precise control, or a limited control. For example, for a soldier fighting game, a game developer may want to give the user or the player all the controls that a real soldier may have in real, or at least the main ones. But for a simple game like pac-man, where there's not much a player can do than just moving around for collecting/eating things, players only have limited controls such as movement in different directions. You can always add more control to your player according to the game.

7.2 Player Movements

A player should have a proper movement. It shouldn't vibrate. It should have a steady movement. The shakiness causes the quality of the game aesthetics to fall and is unpleasant for the eyes. So the sprites must be chosen wisely especially for the animations.

Furthermore, the player movements must be natural and logical. The player or the enemies shouldn't pass through the walls or the floor/ceiling. They shouldn't defy the laws of nature and physics, unless the game asks for it. Never call these bugs as features!

7.3 Camera

Camera is really important when it comes to show the point of view. It is what point of view of the game you want to focus more and it can become important when you want to shift the focus of the player from one thing to another. This obviously isn't the case with board games or maze

games like pacman where the focus is only the board or the maze. You should be very careful when changing the camera movements. For example, if camera screen is being moved to view a different perspective, it must be made sure that the camera movements are smooth and they are not unpleasant and shaky.

8. Objects' interactions

8.1 Collision

Your game starts to take real shape when you add collision within objects. In order to add collision, you can make your own collision function or you can even use SDL's intersection function SDL_IntersectRect that returns true if any of the object's rectangle intersect, otherwise false. If you haven't read Lazy Foo's collision detection lesson, then you must give it a read.

Checking for collision with objects may also increase complexity in a game. Like in a fighting game, you may need to check the collision of every bullet with every hittable object present in your game in every frame. If you have too many objects, then your game would become extremely slow. Rather you should always try to for effective techniques. An effective technique would be to divide the screen into a grid and checking the collision of the player with the object only within the nearest grid box so that bullets currently in first grid in a frame are checked with the elements of first grid only.

Even the sample game pacman uses an efficient technique. Rather than checking collsion of pacman with the food items of the entire grid in each frame, pacman only checks if there is a food item in it's current cell. It's current cell location is updated every time the character moves based on his direction.

8.2 Timed items

Timed items are often called bonus items that are showed on the screen if a player plays well or if a certain time has elapsed. Timed items may include an extra life pack, a sack of money, or any other item that may be important to the player.

IV

Game

9. Designing Levels

9.1 Fixed Levels

Many games we play today have play today have fixed levels, meaning that the level layouts have been predeclared and saved in the code of the game. This is especially common in games that are story based. Implementing this popular approach will require you to design these levels in advance and test them out. Designing such levels would vary from each game, it may be the matter of designing mazes, obstacles, order of enemies or even tasks.
These levels may be saved in the code of the game or as a saved state which will be discussed later on how to do that. If the instructions are saved in the code of the game, it will be hidden from the user and maybe more difficult to alter or add new levels depending on how you do it.
If your game has fixed levels, the advantage may be a linear growth of difficulty as the levels progress but at the same time have this disadvantage that your game will be completed in a limited time, depending on how many levels can you program.

Examples: GTA, Pokemon Nintendo games, Mario, Angry birds

9.2 Random levels

Another approach to designing the layout of your game can be to generate levels randomly. This will most likely be more difficult than simply hard coding a level but will be more rewarding. The sample game provided to you generates levels randomly so it's code can be reviewed to better understand in detail possible ways of applying this approach.
Random level generation doesn't literally mean generating the entire level randomly and being done with it. Rather, along with randomness there should be a set of very intelligent rules that give the game some playable sense. For example, if your game relies on a maze, it should be made sure there are no dead ends of areas that are completely surrounded by walls and therefore inaccessible.

If you generate enemies randomly, and if there are many different levels of enemies, an algorithm should be implemented that give higher probability to the weaker ones as well as keeping check of a reasonable time span between which enemies spawn.
An important advantage of generating random levels is that your game is always different and you can generate all possible levels if you have mastered the random level generation. Unlike having fixed levels, randomly generating a level makes the game a bit more difficult as the user cannot anticipate the next level by simply playing the game again from 0.
Additionally, you may even generate levels randomly but fix their states by testing them all and deciding in their difficulty if that is what you would like to do.

9.3 Endless level

Perhaps most of the addictive mobile games available are based on an endless level scheme. These games do not have explicit stages and therefore do not give the user a sense of completion but rather push the player to continue playing **'endlessly'** to challenge himself and eventually beating only his highscore. These games do not require the additional effort to design multiple levels for the user to continue playing. These games are therefore based all around a scoring mechanism. As the level progresses, the difficulty of the game may implicitly increase to not let the user get bored once he has played enough of the game.

Examples : Flappy Bird, Subway Surfers, Sims city

Whichever method of game designing you choose, it is always a good idea to go through games of similar design to get a better understanding if implementing it, as you'll be learning through the products of professional game developers.

10. User interface

10.1 Screens

One of the most important aspects of a good video game is the user interface (UI) that it provides. A user interface is something by which the user can interact with the computer in order to play the game. Sometimes, some important information about the game needs to be shown to user. Sometimes, the user needs to give input to the computer to do different tasks such as restart, pause or exit the game. For this purpose we use different screens to handle different types of inputs and to show different information.

A game may have many different screens, for example a splash screen, a menu screen, an excerpt screen, a game screen (the main playing screen), a pause screen, and and an exit screen and if necessary, a game over screen to show that the player has lost the game or a winning screen if the levels are not endless. A video game can have more screens depending upon the implementation and the requirement of the game. Since all these objects are all screens just with different purposes and layouts, the best class design would be to create an abstract `Screen` class (because a screen's object will not be created, only its child classes will be declared as as objects in `main.cpp`, and inherit all other types of screens from this base class.

The game provided you as a sample contains the following screens.

10.1.1 Splash Screen

A splash screen is a graphical control element consisting of a window containing an image, a logo, and the current version of the software. A splash screen usually appears while a game or program is launching.

A splash screen is a simple class where only the rendering of the image takes places. It is

shown for a very short span of time after which transition to menu screen takes place.

10.1.2 Menu Screen

Menu screen is an interface that allows the player to choose between options necessary for playing a game. A simple menu screen may consist of a *New Game* button to start a new game, a *Continue Game* button to load a saved game or an *Exit* button to exit the game. It may however also include options for players such as choosing between single player or multiplayer, or choosing the difficulty level etc.

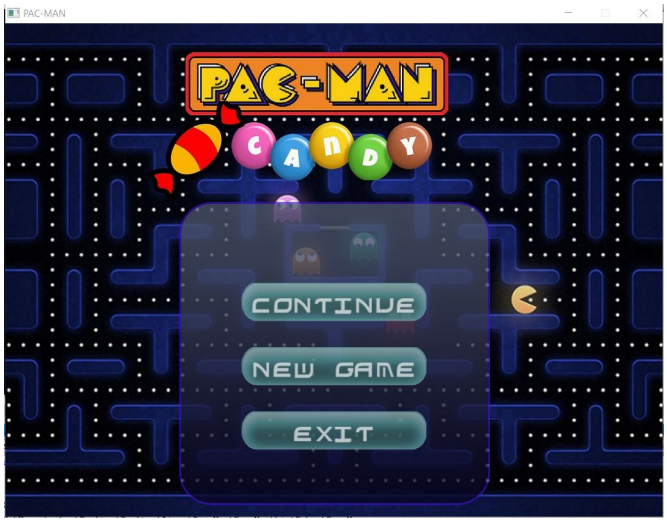


Figure 10.1: Sample Game's Menu Screen

10.1.3 Excerpt Screen

An excerpt screen is used to display any important information that the player should know before playing a game. You can also show the instructions (what the player has to do in that game) or the story of the game in an excerpt screen, or you can make a separate screen for instructions which can be accessed through *Instructions* button in menu screen.

10.1.4 Game Screen

This is the main playing screen. This often contains necessary options that are required while a game is being played such as a *Pause* button. A pause button gives access to pause screen. Some necessary information is also shown on the game screen such as the current score, level number being played or any other information related to the game like lives remaining, status of health bar etc.

10.1.5 Pause Screen

Pause screen is an important screen in a game. A *Pause* button in game screen is used to access this screen. Some necessary functions or buttons that a pause screen should contain are:

Figure 10.2: Sample Game's Excerpt Screen

Figure 10.3: Sample Game's Game Screen

- *Resume* button to continue playing the game.
- *Restart* button to start the game again.
- *Quit* button to give access to quit screen.

Your pause screen may also contain some special functionalities such as options for mute/unmute, or *Control* button to show a screen displaying or changing the controls of the game etc.

10.1.6 Exit Screen

An exit screen or quit screen is usually accessed via *Quit* button in pause screen. It is used to quit the game but with an added functionality of saving a game. Some necessary functions or buttons that an exit screen should contain are:

- *Yes* button to save and then quit.

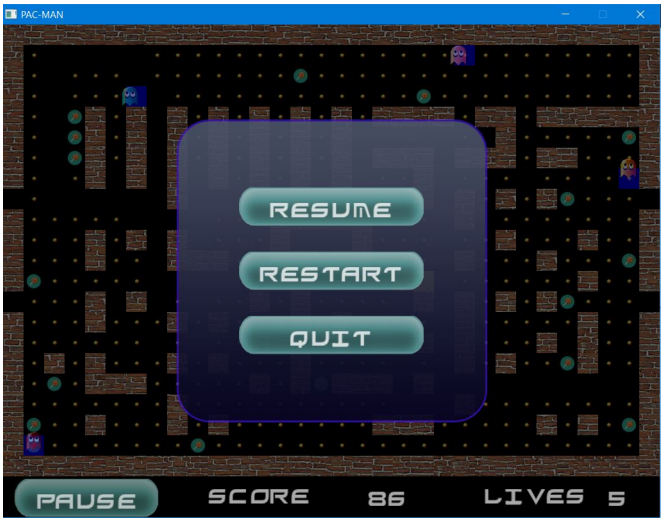


Figure 10.4: Sample Game's Pause Screen

- *No* button to quit without saving.
- *Cancel* button to resume playing the game.

Figure 10.5: Sample Game's Exit Screen

10.1.7 Game Over Screen

This screen is rendered when a player loses the game. It usually shows the highest score of the player, the score of the last game played etc. You can also make a game winning screen similarly if the game does not have endless levels.

10.2 Scoring

In games, score refers to an abstract quantity associated with a player or team. Score is usually measured in the abstract unit of points, and events in the game can raise or lower the score of

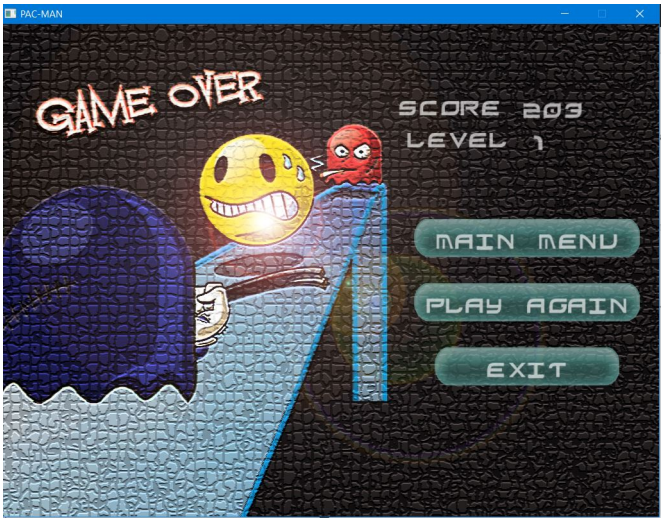


Figure 10.6: Sample Game's Exit Screen

different parties. Most games with score use it as a quantitative indicator of success in the game, and in competitive games, a goal is often made of attaining a better score than one's opponents in order to win.

Usually the score of a game have relevance to game play. In fighting games, for example, scoring a very high number of points could result in unlock-able players or modes or they can unlock the next level like in the sample game. In some games, reaching certain scores gives an extra life, or a continue. All this depends on the type of the game. There is often a time bonus which can add extra points.

In other games, points are typically gained from defeating monsters and enemies. When defeating a boss, a proportionally large number of points is usually rewarded. Extra points can be gained from gathering items, such as power-ups or other pick-ups.

Usually, when a player gets a certain number of points, they may get an extra life or go on to a higher level. Points can be often used as currency which can redeemed for rewards and player upgrades.

11. Save/Load

11.1 Game Save

11.1.1 Saving in a `.txt/.csv` file

One of the easiest ways of saving a game's progress is to save the state of all the variables of the game (that are important to load afterwards) in a `.txt` or `.csv` file. All you need to know is how to write to a `.txt/.csv` file in c++, and save all the necessary information like the player's position, its health status, the score of the game, enemy positions and their health etc. in the file. Use an object's getter functions to save states for them. Below is a function for saving game.

```
void Game::SaveGame()
{
    std::ofstream file;
    file.open("GameData.csv");
    file << /* save any variable's state */ << "\n";
    file.close();
}
```

Listing 11.1: Saving a game

Sometimes, the information to store is often lengthy. Consider the example of the sample game provided to you. Since the levels are randomly generated, saving a game will also require you to save the level. Each level is a 2D array of Cells of dimension 22×32. Each cell contains some information about it, whether it has a wall or any fooditem (candy or lollipop), is it restricted to not have any walls? Storing all these states for all 22×32 cells can be very lengthy. Instead, one should go for easier ways that will require lesser effort to save. Since we need to store some information about each cell, give each information a symbol. For example:

- A cell filled with wall will have a symbol: "|_|"
- A cell which is restricted to have a wall will have: "x"
- A cell containing a candy will have: "*"
- A cell containing a candy will have: "**".

This will make a lot easier for the data to be stored. So the data about the entire maze can saved in a .csv file as: The problems can arise when someone tries to modify this file which can cause

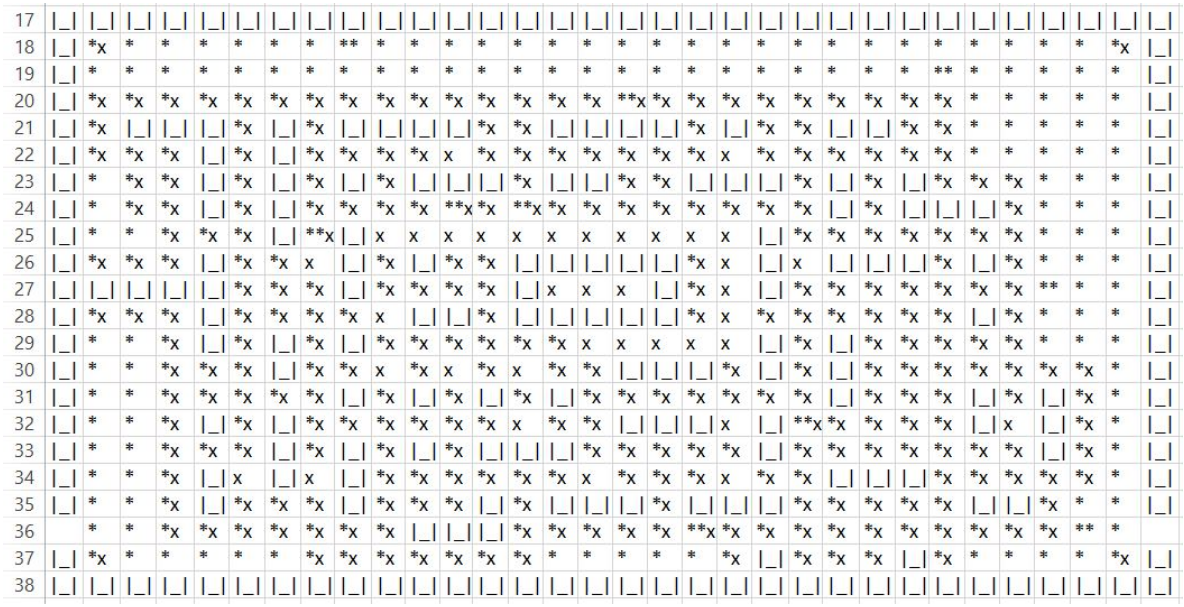


Figure 11.1: Storing information about maze/level

problems. Players can also edit the file to hack the game and customize anything that may help them in winning the game. One way to avoid hacking is to avoid labeling the variable's state that you will save. So that no one knows what the values in the file actually represent.

11.2 Game Load

One saving game has been done, loading a game is not difficult at all. It is only a matter of setting your variable's according to the file data after reading the file previously saved.

```
void Game::LoadGame()
{
    ifstream file ("GameData.csv");
    string line;
    if (file.is_open())
    {
        getline(file, line);
        //load the level number of the game played last time
        level_no = atoi(line.c_str());
        //now close the file
        file.close();
    }
}
```

Listing 11.2: Loading a game