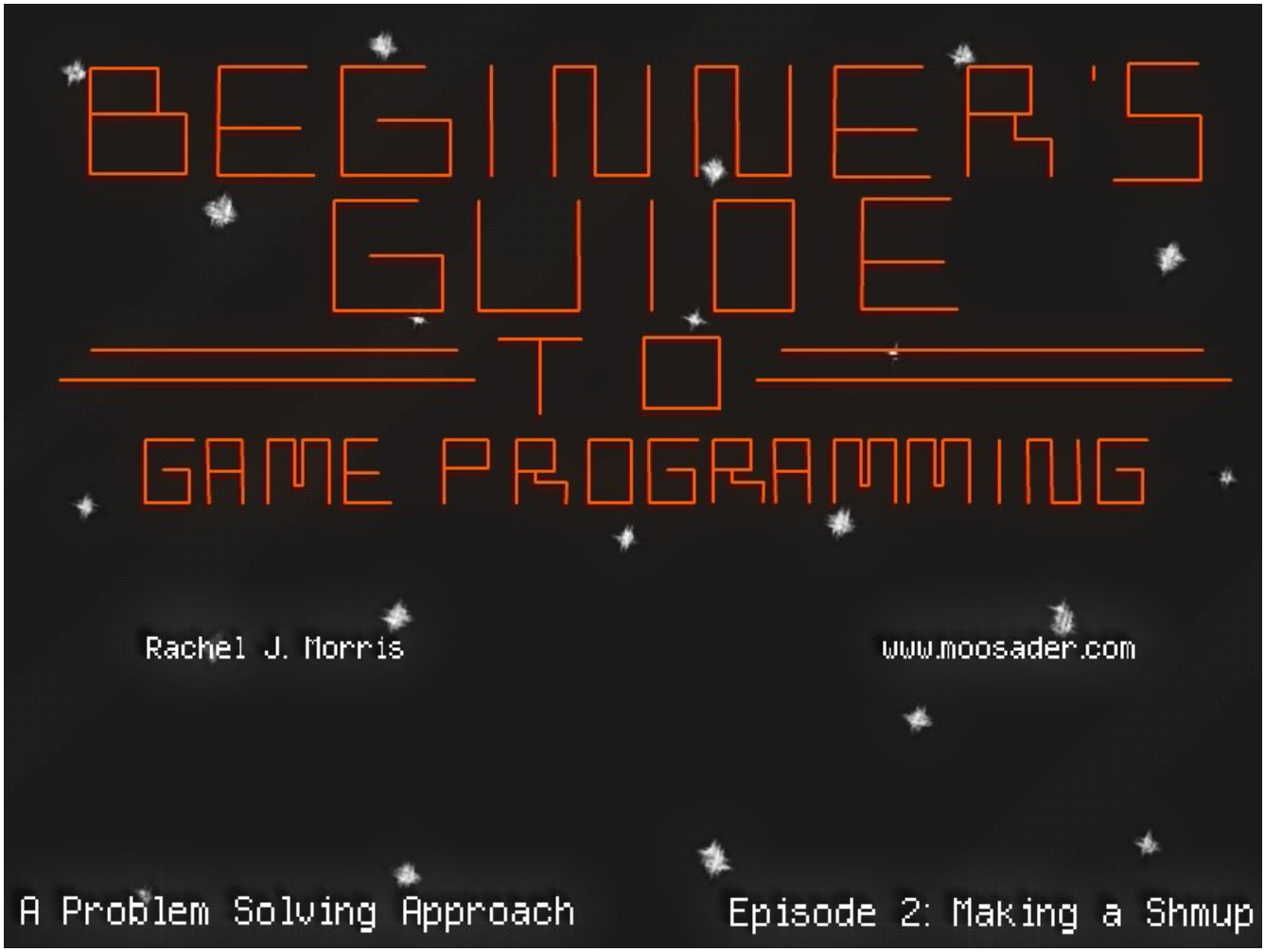


Beginner's Guide to Game Programming: A Problem Solving Approach

Episode 2: Writing your own SHMUP game

v0.1

Tutorial, game, and graphics by Rachel J. Morris

To see a video version of this tutorial, other tutorials,
and check out my open-source C++ games, please go to www.moosader.com

Table of Contents

Revision History

February 9th - 20th, 2010
School is over, example game is done, added to written guide

October 2nd, 2009
Initial writing

Introduction

Welcome back to Beginner's Guide to Game Programming – A problem-solving approach. This is the second in my BgtGP:APSA (whee!) tutorials. While the first episode was to get you familiar with the basics of Allegro and/or SDL, this tutorial is going to be more library-independent.

What is BGtGP: A Problem-Solving Approach?

Beginner's Guide to Game Programming: A Problem-Solving Approach is a series of tutorials that, rather than explicitly giving you all the code to make a game, it focuses on giving you the theory and examples, in order for you to develop one of the most crucial skills in programming – Problem Solving.

There is a sample SHMUP game (or two) on my website, to show you how I would organize and write my code if you get confused at some point, but you're encouraged to go through the guide, and write your own classes and code based on how you think it should be structured.

What are we making?

We are going to make a SHMUP – Shoot 'em up. Games that are Shmups are Gradius, R-Type, and even Galaga. The reason Episode 2 is about Shmups is because it's fairly easy to make a basic one. This tutorial will cover creating basic levels, enemies and players, their bullets, and basic collision-detection.

What should you already know

This tutorial assumes that you know your basics of C++, including:

- Variables
- If Statements and Loops
- Functions
- Classes and Inheritance
- Pointers

It also assumes you know how to do the basics with either Allegro, SDL, or a library of your choice – graphics, input, and sound. The examples in the tutorial will either have Allegro or SDL as an example, or pseudocode, but it should be obvious what is happening when “play_midi” and you should know how to implement this in your own library.

If you're not familiar with a library for handling graphics/input/sound/etc, or if you don't know any C++, then...

Episode 1

Episode 1 of the Beginner's Guide is meant to familiarize you with the basics of Allegro and SDL. If you aren't familiar with a library, you should check this tutorial out, or any other Allegro or SDL tutorials (such as Loomsoft and Lazyfoo, respectively).

You can find Episode 1 by going to www.moosader.com or www.youtube.com/lusikkamage

"Game Development: Where to Begin" tutorial

If you haven't done any game programming and are interested on ***how to get started***, then this is the tutorial for you. There is both a written and video version, and this tutorial covers various languages, as well as engines, and what you should use based on what your experience level is and your goals in game development.

You can find "Game Development: Where to Begin" by going to www.moosader.com or www.youtube.com/lusikkamage

Have additional questions?

If you have questions on portions of the tutorial, you can either comment on my YouTube (www.youtube.com/lusikkamage), email me at racheljmorris@gmail.com, or post on the Moosader message board (http://board.moosader.com)

Lesson 1: Game Design

Design is very important in game development. We are going to start off by planning out what our game is going to have, and what resources we will need, so we can collect them all at once and don't have to go back later to add things in.

Design consists of planning out everything from a theme (space, fantasy, pirates, brocolli), gameplay type, resources you'll need (graphics, sound), and even the structure of your code.

First off, let's start with the creative side. Start brainstorming ideas, and from there have a more specific design, and a list of what your game needs.

Theme

Where will our shmup take place? Tons of shmups take place in outer space, but don't just settle for what everyone else does! What other settings are there? I've seen a shmup where you're a witch on a broomstick, you could be a pirate ship, airplane, rabid hobo, etc. Decide a theme, then decide who the baddies are.

How many levels do you want? Ideally, every few levels should change up scenery, so what sort of environments should there be?

Gameplay

Shmups always have power ups-- multiple types of weapons, usually with different levels. Barriers. Screen bombs. Remember that the baddies also need firepower!

Also, how many bad-guy types do you have? There should be movement types for the baddies (straight forward, sine curve, hone in on player), and different bullet patterns (pea shooter, spreader).

How about the smaller things... What happens when the player dies? Do they lose all their weapons, like in most shmups? Do they have a health bar and can take a few bullets before death? Do they have limited lives?

Resources

It's a good idea, though not always plausible, to try to figure out what graphics, sounds, and other resources you need ahead of time. This game should be relatively easy to plan for. Also keep in mind that this can be a small game! Maybe three levels, five monster, three weapons...

Remember that you can always start out small, and add-on later!

Here's some samples of what graphics you need. For a shmup, really the player and baddies only need graphics for one direction.

Graphics

- Characters (player, baddie)
  - Moving animation loop (you should have a few frames, but if it's a space ship or something there won't be much animation going on. Maybe the jets in the back)
  - Death / explosion animation
  - Animation for when they shoot/attack
- Environment
  - Background image – Space? Sky? Water?
  - You can add more if you want (tiles and such, ground, obstacles), but having maps like in Gradius III would require a more sophisticated map format than what I'm going to cover.
- Bullets
  - Just an image of a bullet. It can be animated or static, depends on what you want. Not much to bullets.
- Powerups
  - Gun A, Gun B, Gun C
    - You should make the powerup kinda shiny and something you want to get. Maybe floats up and down. Or if you want to confuse players, you could make powerups look like death-poop, and then it'd be like “SURPRISE! YOU GET A LAZOR RIFLE!!”
  - Other powerups?
    - 1 Up
    - Point multiplier
    - Speed up/down

        - Barrier
- Title screen
    - Title text
    - Mini-menu (“Play”, “Help”, “Quit”)
    - Any graphics for help menu
- Game over screen
- “You win” screen

Sounds & Music

Yay Sound-effects!

- Explosion / death noises – If you have more than one, then the same sound over and over won't get so dull
- Bullet shooting noise – Preferably different noises for different guns
- Title screen song, songs for different levels, game over song

Lesson 2: Code Design

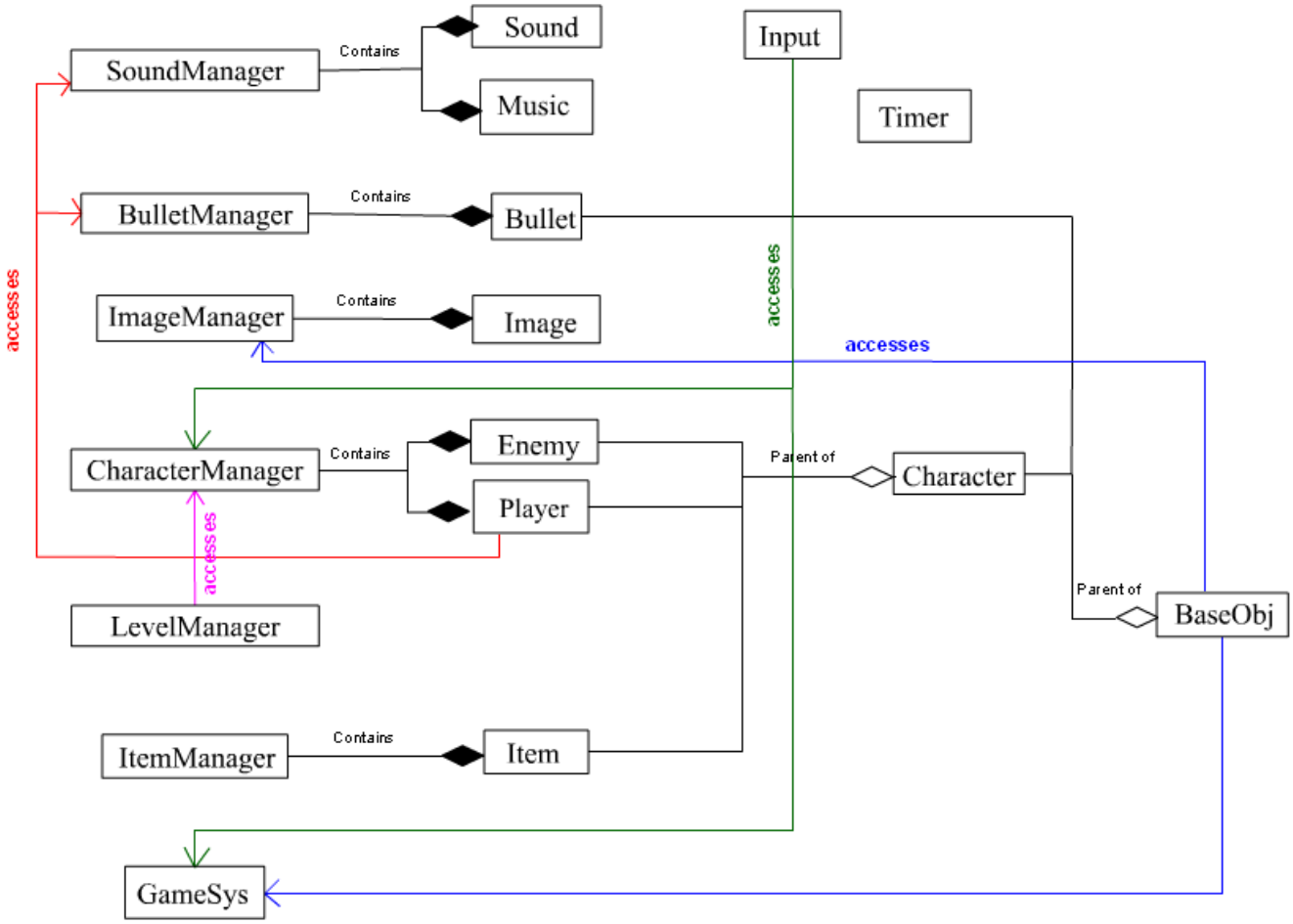
So you know how to draw graphics, play music, and get player input, but what's a good way to put it all together?

We want to minimize repetitiveness (ie, not have duplicate code in multiple places), keep it object-oriented, and hopefully make the code clean enough to read through as well. We should try to make things based around *objects (nouns, what they are)* vs. their *functionality (verbs, what they do)*.

I'm going to talk about a lot of the basic bits you're going to need for the game, but I'm only going to cover how to code specific things. This is your project, and when it comes down to it you make the decisions on how things are written. As such, I don't feel like I should be telling everybody how to write their Player class, but I will cover what some of the basic things you should find, and I will suggest how to organize certain things.

As an example, on the following page is an example of the relationships between my classes:

On the left side of the diagrams, I have my global singleton classes (more on these later). They are managers and contain all of the resources for the game, from the images, to the characters in the game, to the levels.



Unicorn Princess Iteration 1 class diagram -
"Accessor lines" are colored, relational (inheritance/composition) lines are black.
"Accessor lines" for children who access the same classes as an ancestor are left out.

The next column of classes are normal classes that contain the more "obvious" elements of a game- sound, image, different types of entities in the game.

As you can see from the diagram, the Enemy, Player, and Item classes inherit from Character, and the Character and Bullet classes inherit from BaseObj. (Item is inheriting from Character only because it has many similar variables and functions to the Player and Enemy, and I would have had to re-write the same code over if it had inherited from BaseObj).

The Input and Timer classes are pretty much standalone, and are only accessed by main(). Input accesses GameSys and the CharacterManager class, as it will get input like "Toggle Fullscreen" and "Quit", as well as "Move Right" and "Shoot".

Make sure you at least looked at the image. I know all too often I would skim over the side boxes and images in my textbooks and miss useful information. ;) Mainly notice what kind of "objects" I'm using in my version of the game. Give some thought into how you would structure your game. It definitely doesn't have to be like mine.

What are your objects?

In-game objects

The more obvious classes in your game will be things like the player, the enemies, the bullets, etc. Also consider what will have common parents... Player and Enemy tend to be animate objects, bullets and items are inanimate. How will you classify them?

Managers

Other objects will be your managers, which will take care of their own particular items- An Image manager to take care of images, maybe, or a Character manager to take care of the character objects, or maybe just an "in game object" manager to handle the player, enemies, bullets and items together.

Game system

Still other types of objects in your game may be the system class, perhaps handling your screen resolution and initializing your library, and an input class and timer class.

What will my "object" class need?

Here, by "object" I mean the characters and items classes.

First, think of what they have in common: X and Y coordinates, Width and Height attributes. Something to specify what image they use – I would not store the actual image, as many enemies might share the same image and we don't want to load that multiple times. Perhaps have an integer to denote a particular image in an array of images to use. They'll also share Update and Drawing functions. These shared attributes will be used to make a class all the other characters/items inherit from.

Secondly, what will specific classes need that *isn't* shared? For example, enemies will need some form of AI for them to move around and shoot, but player and item classes won't. Enemies and players may have animated sprites and movement speed, but not items (though it can be shared, up to you).

What will my "manager" class need?

The manager will handle all of some object- the ImageManager may have all of the images in the game. This way they can be loaded and deleted at once. In order to draw a specific image, another

object may call the ImageManager with coordinates for it to draw an image at, and maybe a number index to denote what to draw.

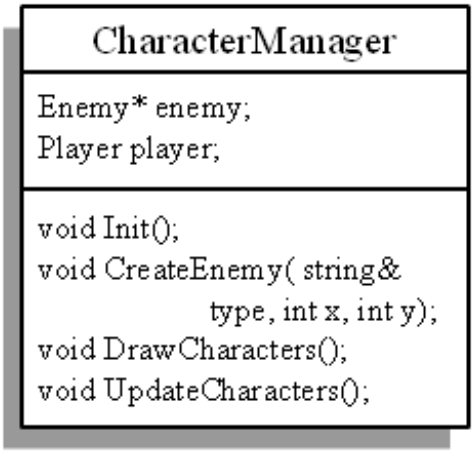
A player/item manager would make sure everything updates and is drawn, and return any necessary info from the player/item objects to the caller object.

A level manager would take care of loading in the levels, and maybe talk to the player/item manager to tell it where the initial positions of each item is.]

Sample classes

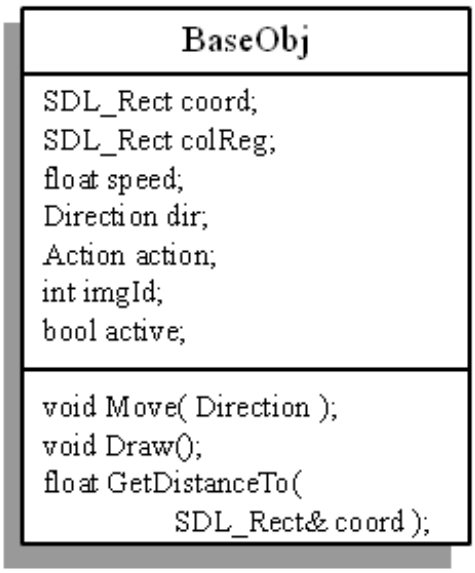
Here are some sample diagrams denoting what my classes store (excluding get/set functions). They also do not include all functions or variables in the classes, but they are to give an idea of what they would contain.

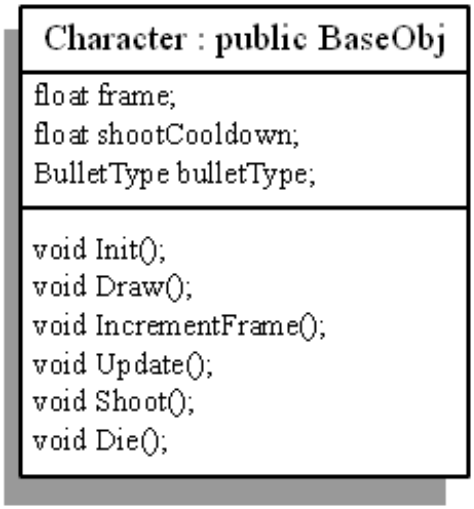
CharacterManager and the Characters



The character manager stores a dynamic array of *Enemy* objects and the one *Player* object. It also stores functions to initialize all the values, as well as creating enemies, drawing the characters, and updating characters.

All the in-game object I have are derived from *BaseObj*:

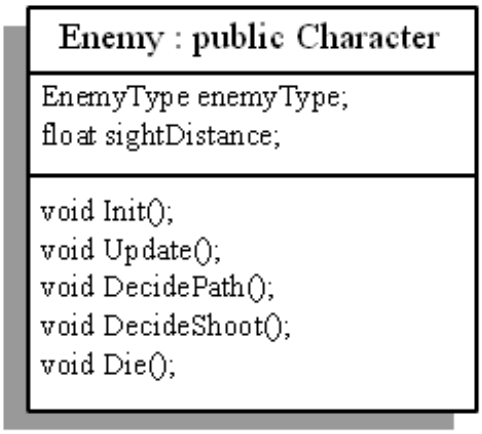


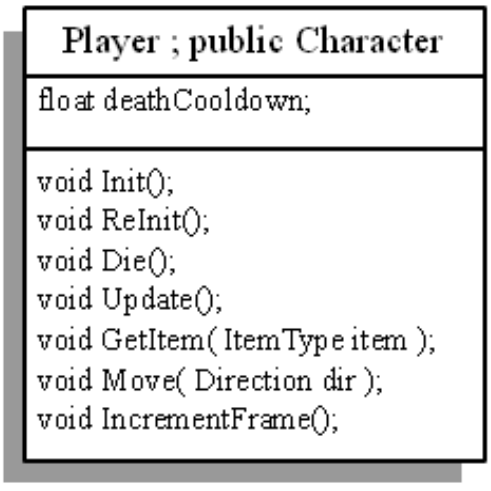


BaseObj mainly holds the coordinates (x, y, w, h) as well as the collision region, described below in part 3. *Direction*, *Action*, and *BulletType* are enumerations (essentially integers with names).

Characters are more than static objects; they animate, so they contain *frame* and *IncrementFrame*, as well as a function for when the enemy or player descendents *Shoot* or *Die*.

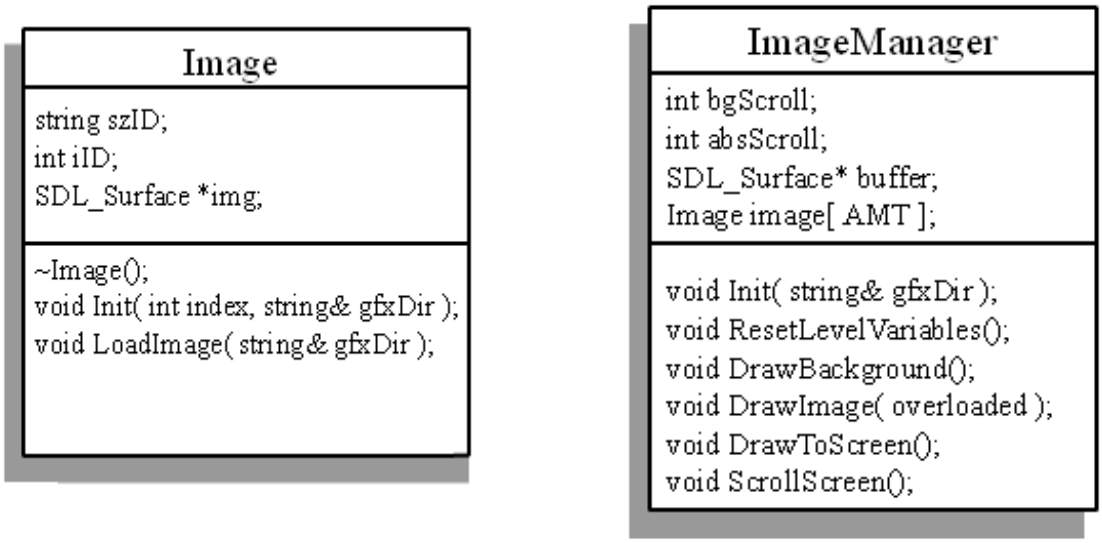
Most of the functions of *Player* and *Enemy* classes are just overwriting functions from *Character...*:





The Enemy class will need to decide how it's moving and decide when to shoot. In my game, it bases this off the EnemyType it is (also an enumeration). If the Player is killed, it will wait for a certain amount of time before respawning, which is the *deathCooldown*.

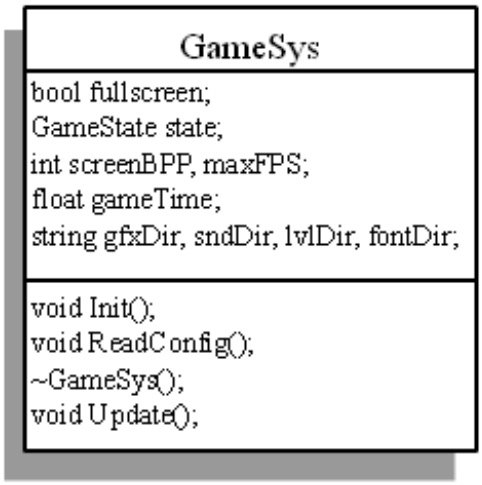
Images and Sound



ImageManager stores the draw buffer, as well as an array of *Image* objects. The *Image* class is optional, I use it to store a String and Integer ID, so that the *ImageManager* can find the image in two ways.

DrawImage in *ImageManager* is overloaded, taking either an integer or string ID, as well as a rectangle storing coordinates, and an optional rectangle storing the crop region.

SoundManager is similar to *ImageManager*, but in my program I had a "*Music*" and "*Sound*" type, as SDL handles them differently.



The last thing I'll show you is my GameSys class. It basically handles initializing (in my case) SDL, as well as reading the config and storing the directories where I store graphics and such.

Is that it?

The “Code Design” listed some general ideas and guidelines to keep in mind while designing your project. It is meant to push you in the right direction on how to create your code, but to not just hand out code you blindly follow (and, therefore, don't quite understand). I am going to give code for the more difficult, specific tasks, but for things as abstract as creating a “Character” class, that's for you to figure out.

This guide is meant to... well, ***guide*** you in programming a game, but not specifically tell you how to do it. The way you structure it is largely up to you, and you are going to need to put some time into designing and planning how it will work together. Even if you're not sure how to start at first, start with *something*. You can refactor later once things become more clear.

The classes listed above *are not* complete classes, they are examples to give you ideas for what could be in your classes. You do not need to use all the same types of classes or managers, it is up to you.

Remember that you are always welcome to post your code at my message board to have it code reviewed by anyone willing and to get constructive critiquing there.

http://board.moosader.com/

Part 3: Specific concepts

Here I'll cover specific things in your game that may be a bit hard to figure out on your own, particularly collision detection and creating a map.

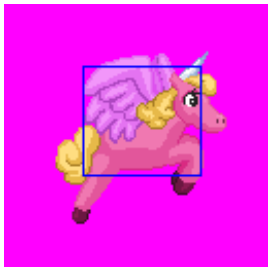
Types of Collision Detection

There are many ways to do collision detection for your game. Some types are using bounding boxes (and, similarly, region bounding boxes) as a quick way to test if enemies are touching, there is pixel-perfect collision detection, which is quite a bit more complex, and comparing distance for collision, where if two entities are within a particular threshold, it's detected as a hit.

For this tutorial, I'm going to explain how region bounding boxes will work.

Region bounding boxes

Generic bounding-box collision detection consists of having a box around the entire image of a character, and testing if the two boxes intersect. Region bounding boxes are similar, except you have a separate rectangle that denotes the "collision region" of an object, based on their image (from (0, 0) to (w, h)).

	
Entire image as a bounding box	Small region as a bounding box

Using a region rather than the entire image is a great idea. It's good practice to keep your sprites in an image with a power-of-two dimension (Unicorn Princess there is in a 128x128 image). It is more efficient, as a computer reads power-of-two's easier, and if it's an odd size, some drivers will append the extra amount of space to an image in order to ***make*** it a power of two. Either way, some people will argue that it's not necessary this day in age with our new-fangled fast computers, but I have seen some lower-end computers crash when given "odd-dimensioned" graphics.

Another thing to point out with the region, is that I didn't have a tight fit around Unicorn Princess. I have enclosed her main body region, and if a bullet flies past her legs, or the back of her tail, it won't really affect her. You can have the region be wherever you see fit, though.

Setting up the collision region

So, how are we going to store this? It might not be as immediately obvious as it sounds, so let's take a quick look.

First off, it's good to store your (x,y) coordinates (and your w,h of the player's image) in a Rectangle object. If you're using SDL, you can use SDL_Rect, but if you're using Allegro you will have to write your own class or struct (I don't know about other libraries).

Second, we will store two Rectangles within the character/item object: Coordinates and what I call “CollisionRegion”. Coordinates are relative to the screen (the player's location), and CollisionRegion is relative to the image (from 0,0 to w,h).

Last, we'll want to make sure the function handling collision will get the correct value for comparing. We won't return just the coordinates ***or*** the collision region, as coordinates is screen-relative but doesn't consider the collision region, and collision region is image-relative and so won't be located at the player's coordinates.

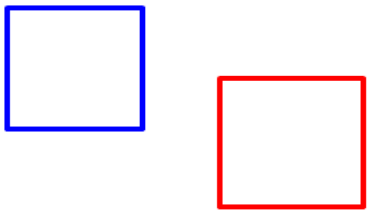
Here's an example of the code for returning the rectangle that we will use to compare objects:

```
SDL_Rect BaseObj::CollisionCoord()
{
     SDL_Rect coordPlusReg;

     if ( active && !dying )
     {
          coordPlusReg = coord;
          coordPlusReg.x += colReg.x;
          coordPlusReg.y += colReg.y;
          coordPlusReg.w -= colReg.w;
          coordPlusReg.h -= colReg.h;
     }

     return coordPlusReg;
}
```

Writing the function to compare

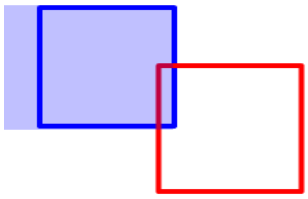
Bounding-box collision detection tends to confuse people. I'm not really sure what the best way to describe it is, but I'll try to make it simple. Sometimes, it just takes some deep concentration and visualizing it in your head for a bit, maybe scribbling on paper to iron out the logic.

The Logic



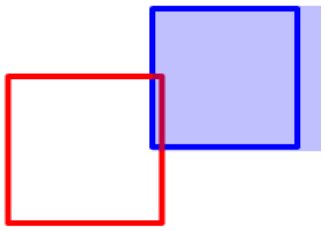
Meet RedObject and BlueObject. Each of them have four sides: Top, bottom, left, and right.
Let's look at one side at a time, ignoring the other sides (since we are comparing them separately).

There is a collision if...:



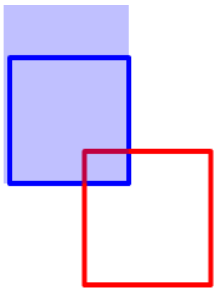
If RedObject's **left** side is anywhere to the **left** of BlueObject's **right** side.

...and...

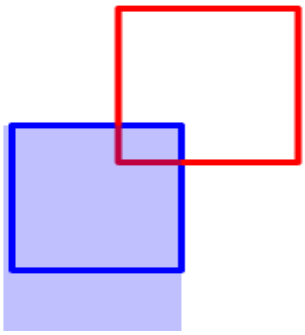


If RedObject's **right** side is anywhere to the **right** of BlueObject's **left** side.

...and...



If RedObject's **top** side is anywhere **above** BlueObject's **bottom** side.
...and...



If RedObject's **bottom** side is anywhere **below** BlueObject's **top** side.

We are essentially testing to see if at least one corner of RedObject is inside of BlueObject. I know this might still be difficult to grasp for some people, but I'm not sure how else to explain it. I struggled with the logic at first, and this is the best way I could explain it to myself. You might check other tutorials to see if they explain it better, but again, I think you should sit down and scribble out what you think the logic is like to make it more tangible.

The Code

Here is my code for the collision function. Make sure you're passing the “Coordinate + CollisionRegion” rectangle (as shown above) when testing it out.

```
bool IsCollision( SDL_Rect& obj1, SDL_Rect& obj2 )
{
    // left1 < right2 && right1 > left2 && top1 < bottom2 && bottom1 > top2
    if (    obj1.x            <   obj2.x + obj2.w &&
            obj1.x + obj1.w   >   obj2.x &&
            obj1.y            <   obj2.y + obj2.h &&
            obj1.y + obj1.h   >   obj2.y )
    {
        // There is a collision
        return true;
    }

    return false;
}
```

The “x” coordinate by itself is the left side of an object, “y” by itself is the top of an object, “x+w” is the right side, and “y+h” is the bottom side.

Creating levels

In this game, we're going to create basic map files with notepad, and load them into the game. This way, they're easily adjustable and you don't have to compile after an edit. It is also to nudge you towards loading in game-specific content instead of hard-coding, and your own basic script.

This level format contains no terrain, but only enemy locations. It's a basic intro to how levels work. More complex levels will be in the episode on how to write a platformer.

Level Format

First you're going to need to plan how your level file will be formatted. Keep in mind that you will run this through a parser in your game to set everything up. Since using ifstream automatically uses white-space as delimiters, I tend to have values separated by that.

Here's a sample of how I specified where an enemy would be in my maps:

```
begin_enemy   type   enemy_00   x 1024          y 192      end_enemy
```

- Here, the “begin_enemy” doesn't really do anything in the code, it's just to have the file itself look a little balanced (for ascetic appeal).
- When the level loader in the program reads “type”, it knows that the very next value will specify (as a sort of temporary value) what the enemy type is. This “type” is used later to store the enemy's other attributes like image and speed.
- When it sees “x”, it knows the next value will be an integer for the enemy's “x” coordinate, etc.
- Those values are stored in temporary variables, and when the parser hits “end_enemy”, it takes all the temporary values and sends them to a function that sets up the enemy's initial (x,y) coordinates, and sets them up based on what the “type” was (for example, set the movement speed, image, types of bullets they shoot).
- My map file also has a few extra values, like “bg_image IMAGENAME” at the beginning so each level can have a different background image.

Loading in your level

Once you have a basic map file written up, you need to write a function to read everything and store the information. I'm going to give my code for a simple parser that reads through the entire file. Like I mention above, it stores everything it reads in temporary variables, and once it hits “end_enemy” it calls a function to store those values as an enemy's attributes.

Here's the basic parser loop:

```
// Temporary variables with default values
// So if something isn't loaded, it has a -1
string temp;
string type = "-1";
int x = -1, y = -1;
bool readFile = true;

while ( infile>>temp && readFile )
{
	if ( temp == "type" )
	{
		// Store the enemy type temporarily
		infile>>type;
	}
	else if ( temp == "x" )
	{
		// Store the x coordinate temporarily
		infile>>x;
	}
	else if ( temp == "y" )
	{
		// Store the x coordinate temporarily
		infile>>y;
	}
	else if ( temp == "end_level" )
	{
		// Stop reading map file
		readFile = false;
	}

	else if ( temp == "next_level_x" )
	{
		// Store the point where it warps to next level
		infile>>warpPoint;
	}

	else if ( temp == "bg_image" )
	{
		// Set the level's background image
		infile>>temp;
		ImageManager::GetInst().SetBackgroundImage( temp );
	}

	else if ( temp == "end_enemy" )
	{
		// Check to see if all values are populated
		if ( type == "-1" || x == -1 || y == -1 )
		{
			cerr<<"Error "<<MISSING_ENEMY_INFO<<": Error loading in enemy,
```

```
                  missing info.  Skipping."<<endl;
            }
            else
            {
                  // Create enemy
                  CharacterManager::GetInst().CreateEnemy( type, x, y );
            }
            // Reset values
            x = y = -1;
            type = "-1";
      }
}
```

When the parser hits “end_enemy”, it checks to make sure all the values were loaded (if not, they are “-1”), and sends it to the CharacterManager's “CreateEnemy” function.

The map format can be as simple or complex as you want, and you can add in terrain yourself with some brainstorming on how it would work. Just start off simple, and add on once you get the easy stuff done. :D

Wrapping up

Hopefully you're feeling up to writing your own SHMUP game now. I've thrown you scraps of a blueprint and some gardening tools, now go build me a skyscraper! :D

Start off with planning and designing (but remember to keep it simple!) then start working on your program, little by little.

Show off your game!

The perfect place to show off your shmup based on this tutorial is at the Moosader message boards! There, you can get your code reviewed, or just get your game critiqued (and played~) by other hobbyist programmers.

http://board.moosader.com/

Please consider donating!

If you have found this or any of my tutorials helpful, or purely enjoy playing my games, please consider donating a little bit so I'll continue creating more educational content!

http://www.moosader.com/

Thank you for your support!

Thanks to...

Thanks to the following people for critiquing this guide!

GroundUpEngine

Kawakami

Protimus