Making Rawr Rinth

Moosader.com / Rachel J. Morris

Work in progress tutorial
last updated January 11th, 2009
Hey, I have full time classes and work, sometimes things need to be put on hold. ;P

Table of Contents

Introduction

What is this tutorial? This is more so a tutorial than a post-mortem, meant to help people figure out how to structure and create a game like Rawr Rinth themselves.

Good Newbie Practice

One thing you might note is that Rawr Rinth is a relatively easy game to make. Because it's not a platformer, there are no formal physics, so one doesn't have to deal with gravity and it complicating the collision detection.

This is a pretty good game to start with because you don't have to worry about physics.

Besides a top-down game, I would say probably the easiest game to start with would be a SHMUP.

Definitely don't start with a "Simple RPG". RPGs require a TON of work, and it'd be insane to try to make one without knowing a scripting language (like Lua or Python) as well.

You have to start simple. You don't have to make pong, and you can make some fun games to begin with, but don't get overzealous!

So what are we using?

We are using C++ and Allegro game programming library.
How do I set up Allegro in DevC++ or Code::Blocks?

Who this is geared towards

This tutorial is geared towards people who are fairly comfortable with C++, and have played with Allegro a bit, but don't know how to structure your games, or where to really begin.

What you need to know

YOU need to know your C++ basics. I'm dead serious. No, stop. Don't give me that whiny "but I wanna make super duper 3D Pokémon MMORPGs now!!!" thing. You need patience if you're going to learn to program.
Here are what I consider the basics, which you can get from any C++ textbook:

- Variables
- If Statements
- Loops
- Functions
- Structs and Classes (mainly classes)
- Inheritance and Composition
- Pointers

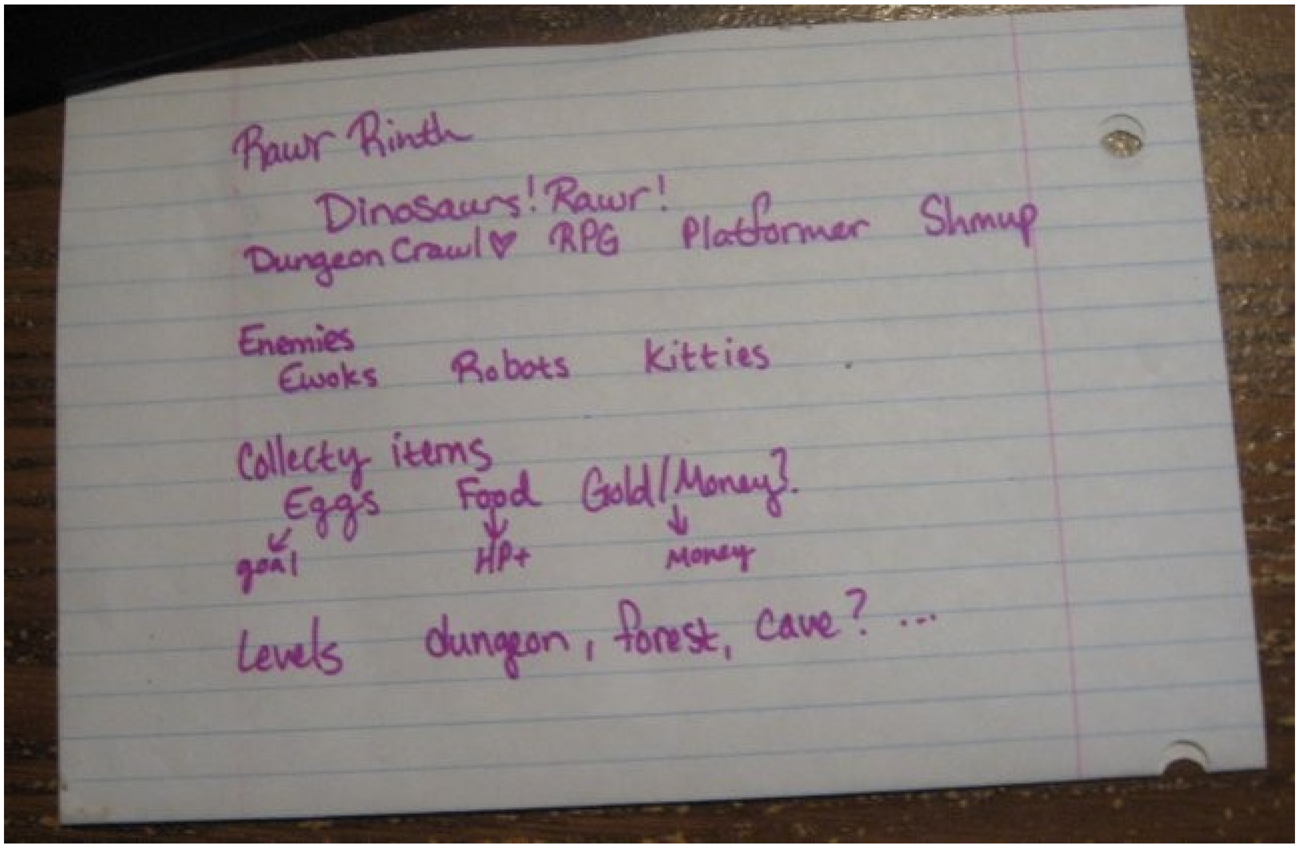
Now with pointers, you don't need to be an expert. Know a little bit about dynamic arrays, and know how to pass pointers to and fro. Both Allegro and SDL use pointers everywhere, and you're going to be a bit confused if you've never seen them before.
Being familiar with Allegro isn't a must, since the functions are generally pretty self-explanatory. If you know SDL, all the better, it's not too hard to interchange these libraries.

Planning the game

Perhaps this is obvious, but not very much planning went into this game. Usually I come up with ideas for things as I am doing the graphics (such as, what is the obligatory collectables you go after for points, what sort of enemies, what sort of characters and such)

Most of what I planned was the gameplay. Initially, I had wanted to make a dungeon crawler, but more action-RPGish like Secret of Mana. Because I had a few weeks to complete it for a class (yay, slacking off), I ended up using an old tileset I had made for houses, so it didn't turn out very dungeon-crawly.

I knew I wanted a dinosaur, and for the most part I just asked friends what dinosuars collect and fight, getting random answers like "Robots and Ewoks".



Rawr Rinth

Dinosaurs! Rawr!

Dungeon Crawl♥ RPG Platformer Shmup

Enemies

Ewoks Robots Kitties .

Collecty items

Eggs Food Gold/Money?.

↙ goal ↓ HP+ ↓ Money

Levels dungeon, forest, cave? ...

pretty standard "design doc" for me. Very professional.

That, my friends, is how you design a mega-awesome game.

Graphics and Levels

Graphics

Okay, we'll get to the programming in a bit. BUT. I HATE working on a game using placeholders. Since I do my game graphics myself, I tend to do as many tiles and characters I can think of, then add on later as I realize what I need.
If you want, you can use the tileset I use in the game, which is actually some of the tiles I did for Elysian Shadows but it's not going to be used so it's public domain now.
Note: Yes, it's a bit annoying that all my tiles are in one long line. I do this because it's how I wrote my map editor to read a tileset. Maybe I will rewrite it to use prettier tilesets one day.

Sound and music

Okay, so you probably don't want to use the music I used, it was just something I threw together in Dr. Petter's Musagi. The sound effects are made with Dr. Petter's sfxr. Both are pretty handy, but are kind of limited to doing old-school-styled music and sound effects.

- Musagi
- sfxr

Levels

The only thing I can think of about the way Rawr Rinth works is mainly how the collision is specified--
In some editors, you can specify areas independently that should be solid, but for Rawr Rinth I just have that "all of (# code for, say, a wall) tile is solid". You can do it either way, doesn't really matter. Collision code still works out about the same.

You can get all the graphics, levels, and sounds for the game by downloading the game.

Programming - Object Basics

So how do we put it all together? If you're like me, one of the hardest parts of writing games is figuring out how everything fits together.
Well, I can't say I know the "right" or best ways to do it, but this is how I have it, and I think I'm on the right track, at the very least.

(If you have feedback, please contact me: racheljmorris at gmail dawt com)

So what makes up a game? Well, you have your **level**, of course.
Levels are made up of **tiles**. Tiles have x and y coordinates, width and height, a number-code to specify which tile it is (for my games, it's the x coordinate on the tilestrip), and probably a boolean storing if it's solid or not.

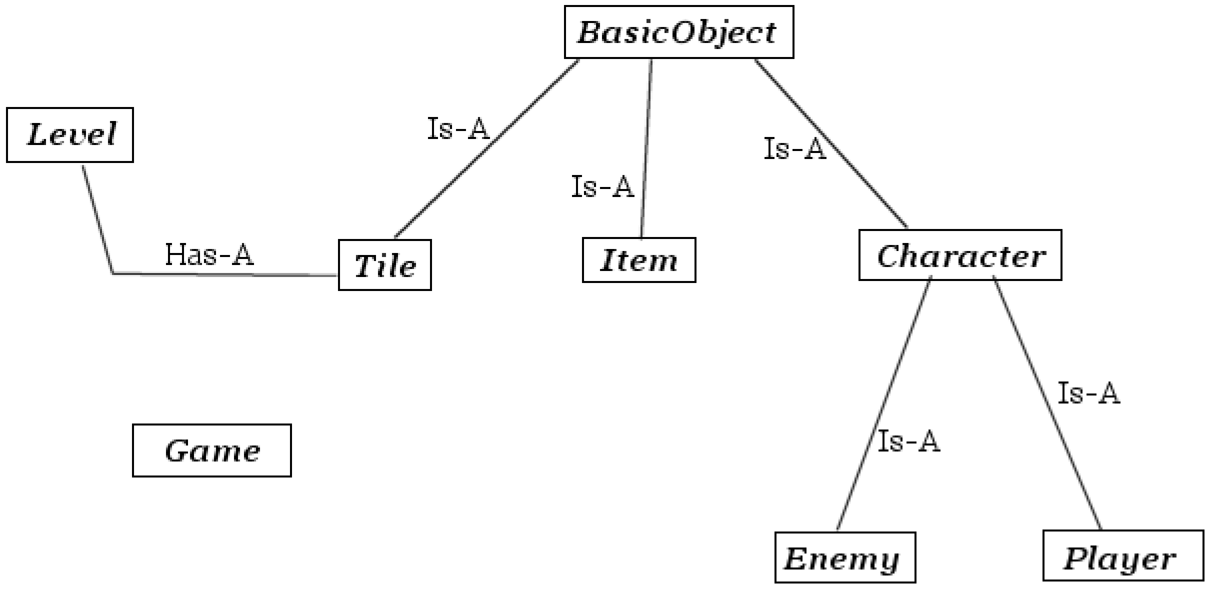
Instead of a boolean that just says if the ENTIRE tile is solid, you might have a "solid region", which would have it's own x, y (based on the tile, not it's location), and width and height.

With SDL, you would use a SDL_Rect type for both the coordinates and the collision region usually.
If you want, you can write your own Rectangle class/struct. Something like this:

```
struct Rectangle
{
  int x1, y1, x2, y2;
}
```

Besides that, you will have your **Player**, **Enemies**, and **Items**.
Although they may not seem terribly similar, they should inherit from the same class.
It would also be good to have the **Tile** class inherit from the same basic object class well, to simplify collision detection.

So, Player, Enemy, Item, and Tile should all inherit from a **BasicObject**, or whatever you wish to name it. Furthermore, I would probably have Player and Enemy inherit from a **Character** class, and then have **Character** inherit from **BasicObject**, as such:



My **Game** object usually stores the initialization code for Allegro, and functions such as ToggleFullscreen that would only clutter up main.

What's in 'em?

BasicObject

BasicObject should just have what Item, Character, and Tile have in common, mainly keeping focus on simplying collision detection. Usually I only have:

```
class BasicObject
{
  protected:
    int x, y, w, h;
  public:
    //Get and Set functions
};
```

Protected acts like **Private**, except that objects that inherit cannot access it's parents private functions, but it will be able to access it's parents protected functions, as it's own private ones.

Good Programming Practice

Yes, keep your variables private (or protected). Write functions to GET and SET the values stored in your private variables.

I will, for instance, have **int x** be private, then overload **void X()** and **int X()** to both set and return the value in x.
You can also have **int GetX()** return what x is, and **void SetX(int val)** set a value to x. I just prefer overloading here.

The reason you do this is to check the integrity of the data being passed!

This means that you want to check to make sure you're not getting a weird value. Maybe someone set the width to -50. You can't have a negative width!

You would check, and if it were not a valid number for that variable, you keep the old value instead of changing, or give some sort of default.

Get in the habit of doing this!

Tile

A **Tile** will be your level's basic building block. In this game, a map is a grid of tiles, each tile taking up a 32x32 space. A tile will have it's x and y coordinates, and you may have width and height variables. Otherwise, you want to have some sort of variable specifying if you can walk on it or not (aka, bool solid).

A tile will only be accessed by the Level class it belongs to. I don't usually worry about having get and set functions for this class, and you can make it a struct if you wish.

Tile attributes usually don't generally during the game, either. You'll load in your map, set the tile's variables, and then just reference them whenever you need to draw the tile.

Level

A **Level** consists of an array of Tile objects, as well as the LoadMap, DrawBottomLayer, and DrawTopLayer functions.

The bottom layers (below the player) have to be separate from the top layers (above the player) because the bottom layer must be drawn first, then the player, then the top layer. Everything is drawn in order, so something drawn first will be below everything else.

Item

An **Item**

Character

A **Character** is an Object that moves. Aside from the basics (**x, y, w, h**), a Character will have attributes such as **speed** (or velocity), **direction**, as well as it's RPG stats (just **Atk** and **HP** for this game). Any of the attributes the Player and the Enemies will have in common should go in Character class.

Player

A **Player** is a "type of" Character. It will inherit from the Character class. The player will have extra stats like **exp** , **level**, **stamina**, **moneyAmt**, and **eggsCollected**, as well as their **score**.

Enemy

A **Enemy** is a "type of" Character and will inherit from the Character class. Enemies will have things such as an **expAmt**, which is the amount of EXP the player gains for killing them, as well as a **type**.

There will also be some AI like **behavior** (Flee from player, chase player, act indifferent). The behavior will merely be an enumerated variable (think of a number- code symbolizing each type), and an Update function will move the character based on what it's behavior type with, with a simple if statement. More on this later.

Game

A **Game**

Under Construction ...