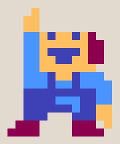


Arid Arnold Bible

Foreword: What is the Arid Arnold Bible? 2
Chapter 1: The Abstract 4
Chapter 2: Code Structure 6
Chapter 3: Entities 12
Collisions 14
Chapter 4: Graphics 20
Camera 20
Rendering a frame 23
Animations 25
Geometry 26
Layouts & UI 27
Effects & Particles 28
Chapter 5: Audio 32
Threading troubles 33
Chapter 6: Creating content 35
Creating levels 37
NPCs 39
Text & Strings 39
Chapter 7: Arcade Games 42
Chapter 8: Final thoughts 44
Further Reading 46

Foreword: What is the “Arid Arnold” Bible?

Online tutorials are a great way to learn the craft of game development. Seemingly, it is possible to learn anything from them: how to do 2D collisions, sprite rendering, sound effects, ragdoll physics, post processing effects, and even more complicated stuff like shaders. Yet, despite this vast wealth of knowledge, there is usually one thing missing: *context*.

If you were to follow a tutorial on how to make a simple button class, you might learn about the hover state, and the focus state. Perhaps it also has a callback OnClick() function, which can be passed in optionally. All good stuff, but what you don’t learn is how this button class fits in within the larger context of your program. A naive implementation would be to just instantiate it in your main program file. This would quickly become a nightmare when you consider that some games contain thousands of different buttons. Trying to position, show/hide, and update all these buttons in one file is untenable. Clearly, these buttons need a context to live in. Thus comes the need for a UI system which can do just that. Then realise that this UI system needs to interface with the rest of the program. For example, a shopkeeper, which is an entity in the game world, needs to open a menu where the player can buy items. The context for the button, the UI system, is itself within a larger context. In fact, the only way to fully understand this idea of context is to understand the program as a whole. That’s what this document is: a full dissection of my latest game “Arid Arnold”.

This idea of context is, without a doubt, the hardest part of programming. In all of my years, I have yet to see a codebase which has perfectly contextualised its component parts. In the beginning it is easy to have clean concepts, clean implementations and simple abstractions. But eventually there comes an exception to the clean concept, then another, and then even more. Until the code base becomes the usual fragile mess of interdependencies, “abstractions” that require intricate knowledge of the underlying system, and functions that do 100 different things. Yes, that includes the project I am about to dissect.

It’s perhaps then not surprising that no tutorial has adequately covered this topic, and I will not either; the “Arid Arnold” codebase is too flawed to be exemplary in this way. Yet, I would still consider it to be successful. Of all the codebases cast aside, scrapped because they were unable to pay their technical debt or worse, “Arid Arnold” stands in the exclusive club of codebases that made it across the line. Released on Sep 22nd 2024, “Arid Arnold” is a 2D puzzle-platformer with 9 worlds, and over 120 levels. It has unique mechanics in each world, all made possible by the architecture I developed.

Initially the game was planned to be a very basic platformer, with only a handful of levels. The programming started with that goal in mind. Fast and simple solutions were the priority. Then at some point I thought it would be fun to flip the game’s gravity, and this gave me the idea to make separate worlds. From there the project grew and grew, into the 9 worlds I ended up with. If I were to start again I probably would have laid the foundations differently. At the end we will discuss this, but for now take this document as merely a description, not a recommendation per se.

Chapter 1: The Abstract

Before we dive into the code let's talk about one level above that. Yes, even the code itself lives within its own larger context. It's turtles all the way down. "Arid Arnold" was developed using C# on top of the MonoGame framework[1]. This barebones framework only provides a few basic functionalities to the programmer:

- Loading data using the content manager
- Drawing using the sprite batcher
- Receive inputs from keyboard or controller
- Play music and sound effects

The rest is up to the programmer. I imagine it like a Super Nintendo, without a cartridge it does nothing when you turn it on. It needs some data which tells the Super Nintendo what to draw, and which sounds to play. MonoGame is the Super Nintendo, our program is the cartridge. We do things internally within our program, then send things over to MonoGame: the magic box that connects us to the user.

So let's put finger to keyboard and write some code, right? Well before you write any code you should always think about **code style**. These are the conventions of how you write your code, both in terms of meaningless syntax aesthetics, but also which language features you want to use. The aim here is consistency. Yes, despite having written the code entirely by myself, it's important it remains readable. I will forget code, and you will too. So when I come back to an object months after making it, I should be able to get up to speed as quickly as possible. Consistency means I can see something and know exactly what it is.

I won't bore you with the laborious details of my code style; those will become evident as we read the code, but here are some key points:

- All member variables must be named with an "m" prefix. Known as hungarian notation. E.g. "mMyVariable"
- All constants must be named in all caps. E.g. PHYSICS_STEPS
- Files must be logically ordered: constants, members, initialise, update, draw, utils
- Use structs instead of tuples
- All members of a struct must be public
- No members of a class can be public
- Use inheritance for all "is a" relationships. E.g. Arnold **is a** PlatformingEntity
- Do not use properties, use functions for getter/setters
- Do not use interfaces, use abstract classes instead
- Do not use lambda functions
- Do not use optional classes, use null instead. You can use "?" on a struct.
- Do not use "var", manually specify types.
- Avoid features I didn't know existed at the start of the project.

Some of these may strike veteran C# developers as bizarre, but understand that I come from a C++ background. A lot of these, such as using public members in structs, not using interfaces, or not using properties mimic how things are done in C++ land. I had realised that some of these were mistakes, yet consistency was the goal. If I change halfway through, half of my classes will use the new style, and half will use the old style. That's no good. So I stubbornly stuck to these rules all the way through. Yes, I will discuss what I would do differently at the end.

I do suggest you play the game a little bit before reading this. Understanding what the code's final output will help contextualise what I'm saying. If I start talking about a "LevelSequence", it will help to have

[1] MonoGame - https://monogame.net/

played the game to know intuitively how the levels are structured for the user. The game is free to download: https://icefish-software.itch.io/arid-arnold

I also invite you to download the source code and read along the parts that I am describing. You can find the code on github, with instructions on how to get it building: https://github.com/AugsEU/arid-arnold.

IMPORTANT: If you want to follow along you might be confused as to why there are multiple files with the same name. For now, just ignore any copy of a file in the “Arcade” folder. This will be explained later.

Much of the code in this document will be abbreviated, so it is best that you do read the full code yourself. I am mostly just covering how the systems work, not the individual things within that system. E.g. I will describe how entities work in general, but not the specifics of how Arnold himself works.

Chapter 2: Code Structure

Enough faffing, enough yapping, let's get to the code. What better place to start than the singleton:

```csharp
/// <summary>
/// Simple singleton implementation. It uses lazy initialisation.
/// </summary>
/// <typeparam name="TClass">Use CRTP to make singleton of yourself</typeparam>
abstract class Singleton<TClass> where TClass : class, new()
{
    protected Singleton()
    {
    }

    public static TClass I { get { return Nested.instance; } }

    private class Nested
    {
        // Explicit static constructor to tell C# compiler
        // not to mark type as beforefieldinit
        static Nested()
        {
        }

        internal static readonly TClass instance = new TClass();
    }
}
```

Here we are defining a class which has a property "I", which stands for instance. The idea is that we have one, and only one, global instance of this class. The code here is a bit weird, but just remember that anything that inherits from it is a singleton. This is similar to a static class, although it holds the advantages of not having to type "static" everywhere, and the possibility to swap out the instance should it be necessary.

The singleton reached its height in the early 2000s, yet in recent years it has suffered significant slander. So much so that it has been labelled as an anti-pattern, and is generally frowned upon by the object-oriented community. It flies against C#'s concept that everything is an object, and effectively allows us to write procedural[2] code. Similar to how in C++ we can write code outside of an object.

Being that only one instance of this object exists, it effectively operates in the same way as static data in C++. But, unlike global variables, it also retains the benefits of being encapsulated, none of the member variables are exposed directly.

[2] Procedural code refers to languages like C, which are not object-oriented. Instead everything is performed by functions or procedures that live outside of objects. Not to be confused with procedurally generated code.

In “Arid Arnold” we use these for their great convenience. The above abstract class allows us to define a singleton using the **curiously recurring template pattern**[3]:

```
class CampaignManager : Singleton<CampaignManager>
{
    // Code here //
}
```

It’s a bit strange but this creates a campaign manager which is a Singleton. We can then access the campaign manager easily from anywhere in the program. Such as in the UI to draw the life counter:

```
public override void Draw(DrawInfo info)
{
    // Get life counter info
    bool isActive = CampaignManager.I.CanLoseLives();
    int currLives = CampaignManager.I.GetLives();
    int maxLives = CampaignManager.I.GetMaxLives();

    // Draw life counter //
}
```

This should illustrate the key benefits of the singleton. There exists somewhere in memory the number I want. An OOP purist would have to jump through several hoops by passing around a reference to the CampaignManager through various constructors so we could get access. With a singleton I can just *get* that value, and why shouldn’t I? It’s right there in memory! An assembly programmer would just load it directly too, and I’ll be damned if my life should be harder than theirs.

The singleton classes are the pillars of the codebase. Every class is managed by one of the singletons, and if you want to get something done, you call the singleton, and it will call down to its children.

Let’s go over all the singletons that exist in “Arid Arnold”, from most important to least:

- **ScreenManager** - From here we can set what screen we are on. For example the main menu is a screen, as is the main game. Each screen essentially acts as an independent project.
- **MonoData** - This is where we load data from. You can call the various Load() functions to load in assets. It also supports path remapping, so you can redirect the “WallTile” texture to custom variations depending on the level’s theme.
- **MonoDraw** - A static class that has many drawing utilities.
- **InputManager** - Handles taking inputs in an abstract way. We define certain actions such as “ArnoldJump”, and the InputManager class is in charge of triggering those when certain buttons are pressed.
- **EntityManager** - This manages all the moving actors you see on screen. Arnold himself is an Entity, as are the coins, and gravity orbs. It is in charge of updating them, drawing them and resolving collisions between the entities.
- **TileManager** - Holds the game’s tiles and draws out the tilemap. This is what is used for the level geometry.
- **CampaignManager** - In charge of managing Arnold’s progress throughout the campaign. For example, which level are we on, which one is coming next, or how many lives we have left. This is one of those singletons that could probably be split up since it is somewhat of a mega class that handles many responsibilities.
- **LanguageManager** - Handles all the strings in the game. It is important to not program string literals like “Hello Arnold” into the game, instead we use a layer of indirection and deal only in IDs.

[3] Curiously recurring template pattern - https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

So instead we would have an ID like “Arnold.Greeting”, which the LanguageManager will transform into what the user sees. This way we can easily localise the game without changing any code.

- **FXManager** - This controls the various visual effects and details that show up throughout the game. An effect(or FX) is simply a thing that is visible, and doesn’t interact with anything in the game.
- **ParticleManager** - Similar to FXManager, but is made for drawing many small particles quickly. The various optimisations necessary for this made it impossible for each particle to be its own FX. So it had to be a separate singleton.
- **OptionsManager** - In charge of the game’s settings. We can easily access things like the sound effect volume.
- **MusicManager** - We can request music to be played and this will play it. It can also fade between two tracks.
- **SFXManager** - We can request sounds to be played and this will play it. It also supports spacial audio.
- **EventManager** - Raises various signals which then trigger certain methods to be called. For example, raising “PlayerDead” will cause the game screen to reload the level.
- **FontManager** - Handles loading the various fonts in the game. This should have been merged into MonoData but never was.
- **FlagsManager** - A bucket that can hold many flags in on or off states. An NPC might set the “kGatewayKey” flag to notify that Arnold holds the key to open the great gate. Games like Elden Ring have a similar system.
- **SaveManager** - Manages saving and loading both settings and campaign progress. This sets the state of all the other singletons.
- **GhostManager** - In charge of drawing Arnold’s ghost image. This is like in a racing game when you can see your personal best as a ghost.
- **ItemManager** - In charge of holding Arnold’s power item.
- **CollectableManager** - Holds how much of each collectible we have. For example, how many water bottles do we own?
- **TimeZoneManager** - Controls the time that Arnold exists in. When we travel to the future this keeps track of that.
- **RandomManager** - Holds two random number generators for convenience. We can probe them to get random numbers. One is for anything that affects the game’s state, and the other is for anything that affects the game’s visuals. The idea being that if someone were to turn up the particle effects, we don’t want that affecting the game’s state. The two are kept separate.

At the top of this list is the ScreenManager, each “Screen” can be thought of as its own scene. They act independently, meaning I can do things in the “MainMenu” screen and that won’t affect what’s happening in the “GameScreen”, where all the action happens. There is only one screen that is marked as the active screen at any given time. So switching between the game and the main menu is a matter of changing which screen is active.

I invite you now to read through the abstract “Screen” class found in “ScreenManager.cs”. Here it is abbreviated:

```
abstract class Screen
{
    public const int SCREEN_WIDTH = 960;
    public const int SCREEN_HEIGHT = 540;
    public static Rectangle SCREEN_RECTANGLE = new Rectangle(0,0, SCREEN_WIDTH, SCREEN_HEIGHT);

    protected GraphicsDeviceManager mGraphics;
    protected RenderTarget2D mScreenTarget;
    /// <summary>
    /// Update the screen
    /// </summary>
    /// <param name="gameTime">Frame time</param>
    public abstract void Update(GameTime gameTime);


    /// <summary>
    /// Draw the screen to a render target
    /// </summary>
    /// <param name="info">Info needed to draw</param>
    /// <returns>Render target with the screen drawn on</returns>
    public abstract RenderTarget2D DrawToRenderTarget(DrawInfo info);
}
```

Each screen has a fixed resolution of 960x540. This was chosen because it is exactly half of the common resolution 1920x1080. This allows the pixel art to be scaled nicely by exactly two. On each frame, we ask the screen to Update its state, then we ask it to draw to a “RenderTarget2D”. A render target is a texture that we draw to at runtime, instead of loading from a file. Think of this like a blank canvas that the program can paint onto. Once the screen has filled in this canvas, it is passed on to the Main.cs file. This can then paint the canvas itself to the screen as if it were a regular texture. We can scale it up if the window is big enough.

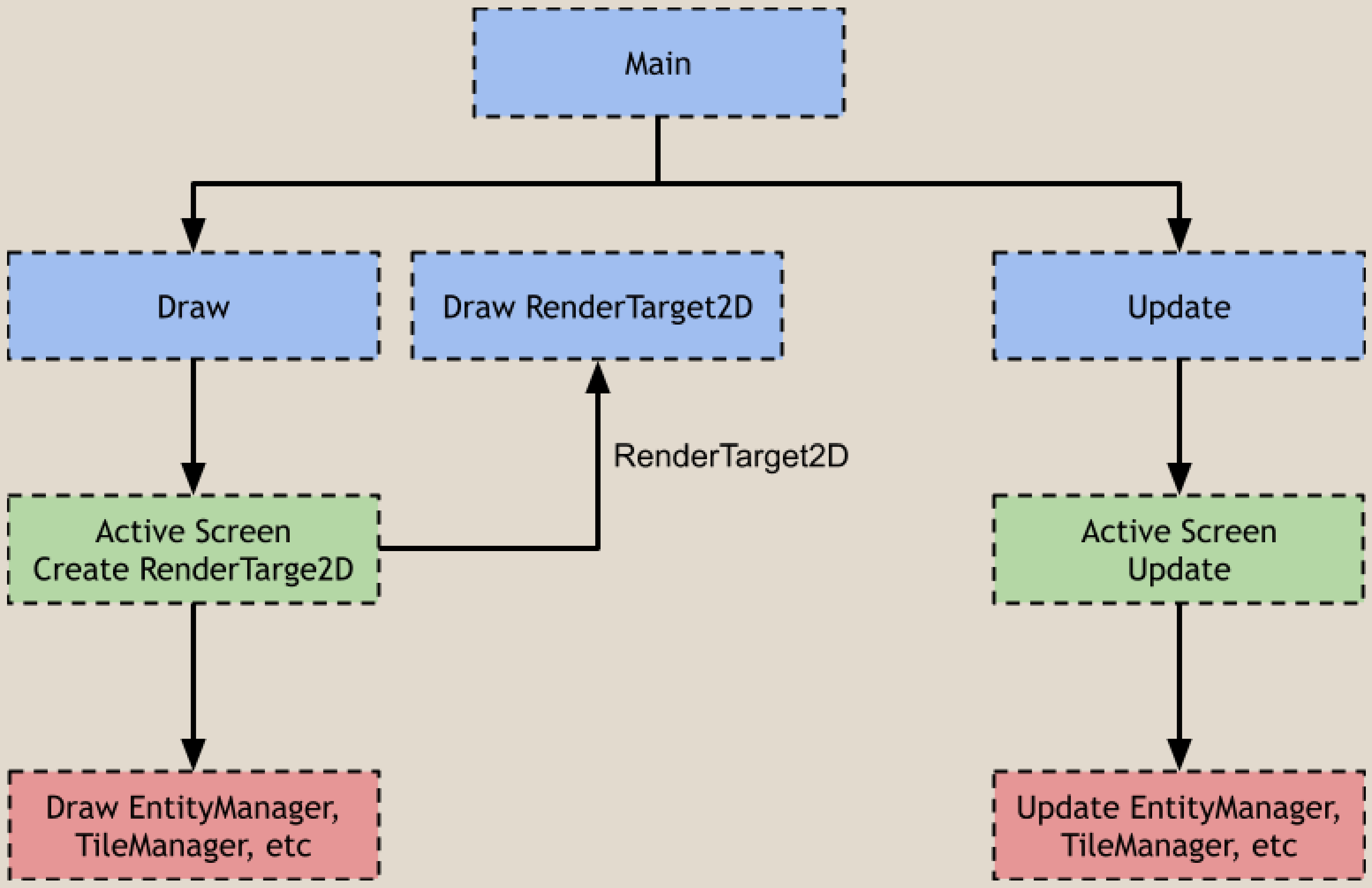
```
protected override void Draw(GameTime gameTime)
{
    //Draw active screen.
    Screen screen = ScreenManager.I.GetActiveScreen();

    if (screen != null)
    {
        // Get the screen's canvas.
        RenderTarget2D finalFrame = screen.DrawToRenderTarget(frameInfo);

        /* Code */

        // Start sprite batch
        screenCam.StartSpriteBatch(frameInfo, new Vector2(screenRect.X, screenRect.Y), null,
Color.Black);

        // Draw the canvas given to us by the screen, scaled.
        Rectangle destRect = GetFinalDrawDest(frameInfo, finalFrame);
        MonoDraw.DrawTexture(frameInfo, finalFrame, destRect);

        screenCam.EndSpriteBatch(frameInfo);
    }

    base.Draw(gameTime);
}
```

The most important screen is GameScreen.cs, this is where the actual game is played, and will be the focus of most of this document. Once this screen is activated, it is responsible for updating the singletons mentioned above:

```
/// <summary>
/// Update gameplay elements by 1 step
/// </summary>
void GameUpdateStep(GameTime gameTime, Level level)
{
    HandleInput();
    EntityManager.I.Update(gameTime);
    TileManager.I.Update(gameTime);
    ItemManager.I.Update(gameTime);
    FlagsManager.I.Update(gameTime);

    level.Update(gameTime);

    EventManager.I.Update(gameTime);
}
```

This is the main meat of the update loop. These are the singletons which encapsulate the game's state. Other singletons like FXManager do not affect the game's state, and so are updated elsewhere. Note that not every singleton needs updating. The singletons themselves then propagate the update call down to their children. E.g. an Entity is updated from EntityManager.I.Update()

In summary, the structure looks like this:


Main
Draw
Draw RenderTarget2D
Update
RenderTarget2D
Active Screen Create RenderTarge2D
Active Screen Update
Draw EntityManager, TileManager, etc
Update EntityManager, TileManager, etc

Since the GameScreen is the most important screen, let's drill down into the flow of that. Firstly let's understand that multiple things can stop the game from actually updating its state.

At the top level, the GameScreen’s update looks like:

```
/// <summary>
/// Update the main gameplay screen
/// </summary>
/// <param name="gameTime">Frame time</param>
public override void Update(GameTime gameTime)
{
    // Update various timers and UI.
    // These always update no matter what.
    mFadeInFx.Update(gameTime);
    mLevelEndTimer.Update(gameTime);
    mMainUI.Update(gameTime);

    // Check if a load sequence is pending.
    if (mLoadSequence is null)
    {
        CheckForLoadSequence();
    }

    if (mLoadSequence is not null)
    {
        // Do load sequence until it is finished
        mLoadSequence.Update(gameTime);

        if (mLoadSequence.Finished())
        {
            mLoadSequence = null;
        }
    }
    else if(mPauseMenu.IsOpen())
    {
        // Do pause menu until it is closed
        mPauseMenu.Update(gameTime);
    }
    else
    {
        // Actually run the game.
        GameUpdate(gameTime);
    }
}
```

There’s two things which can stop us from updating the game, a loading sequence or the pause menu. Once we know neither of those is happening we start to actually update the game’s state.

GameUpdate() updates things that are in-game elements. But this doesn’t mean we always update the entities in the game. The camera can block the game’s update. This happens when we do stuff like rotate the stage, we want to pause the action so the player doesn’t get disorientated. Also, when we win a level we pause the game while we display the “Level Completed” text. If all is normal we then finally get to update the game’s entities in GameUpdateStep(). Now that is understood we can move on to how these entities work.

Chapter 3: Entities

In “Arid Arnold” the concept of an entity is more limited than in game engines like Unity; it is not the case that everything is an entity. Rather, entities in “Arid Arnold” are just the moving actors. This includes Arnold himself, NPCs, enemies, bullets, etc. Let’s look at an outline of the key elements of an Entity:

```
abstract class Entity
{
    protected Vector2 mPosition;
    protected Texture2D mTexture;
    protected float mUpdateOrder;
    private bool mEnabled;
    private InteractionLayer mLayer;

    /// <summary>
    /// Get the collider for this entity
    /// </summary>
    /// <returns></returns>
    public virtual Rect2f ColliderBounds()
    {
        return new Rect2f(mPosition, mTexture);
    }
}
```

Every entity has a place in the world, stored as a member, and a collider made from a virtual method. Some entities can then override ColliderBounds() to have a different one, but they are all AABB(axis-aligned bounding boxes). This is much more specific than the traditional “GameObject”, but it allows us to make useful assumptions. For example we can easily check if two entities are touching, then inform them so that they can interact. This is done in the EntityManager:

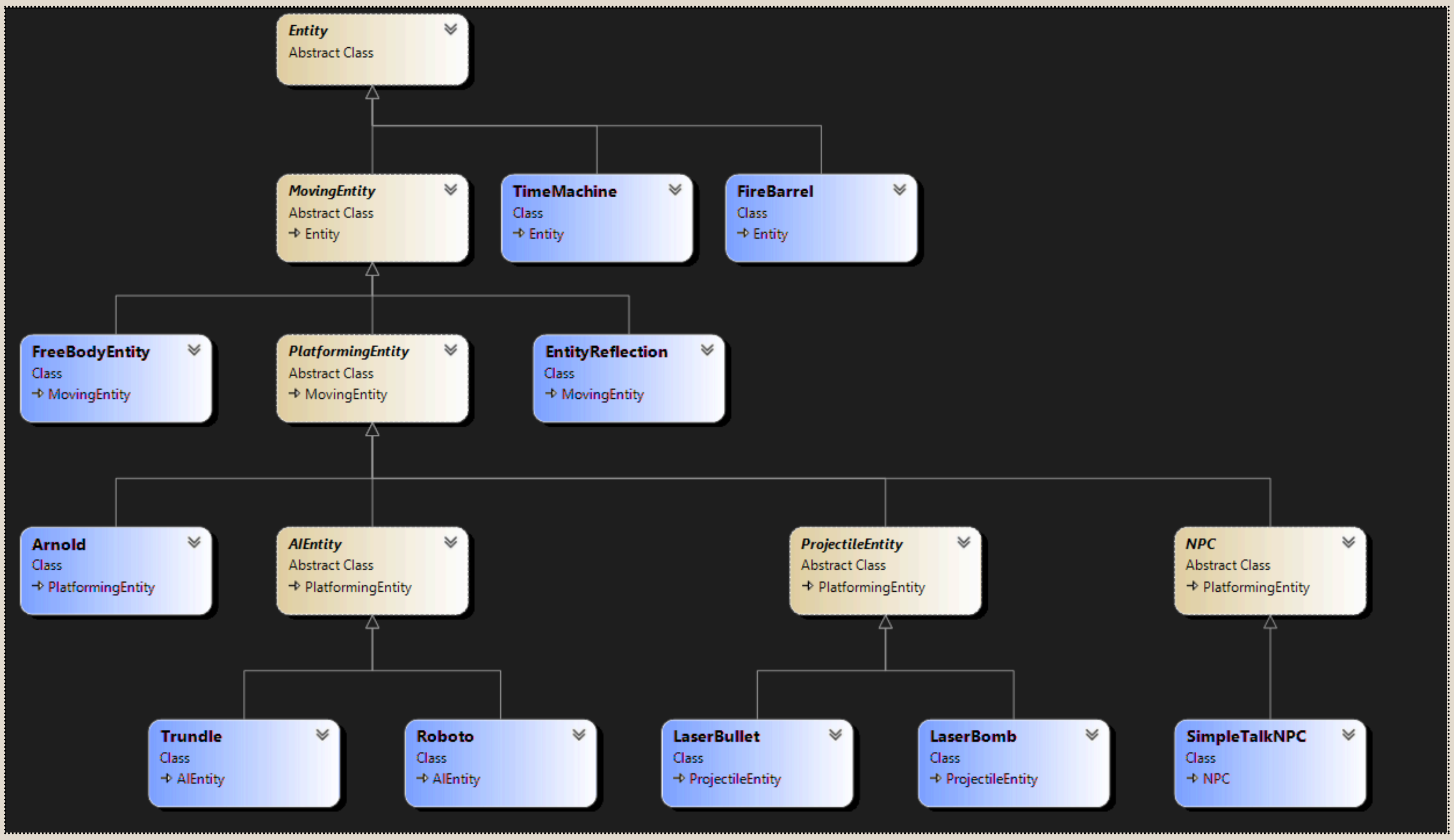
```
void ResolveEntityTouching()
{
    // Not the best but the number of entities is small enough for optimisations to not be needed.
    for (int i = 0; i < mRegisteredEntities.Count - 1; i++)
    {
        Entity iEntity = mRegisteredEntities[i];
        if (!iEntity.IsEnabled()) continue;

        Rect2f iRect = iEntity.ColliderBounds();

        for (int j = i + 1; j < mRegisteredEntities.Count; j++)
        {
            Entity jEntity = mRegisteredEntities[j];
            if (!jEntity.IsEnabled()) continue;

            Rect2f jRect = jEntity.ColliderBounds();

            if (Collision2D.BoxVsBox(iRect, jRect))
            {
                //Both react.
                iEntity.OnCollideEntity(jEntity);
                jEntity.OnCollideEntity(iEntity);
            }
        }
    }
}
```

This isn't the most efficient way to do it, it is an O(n^2) algorithm when we could use a quadtree to make this O(n). But we get away with it since there aren't usually many entities on screen at a time. There's no need to complicate things unless necessary. So for example, this is how a bullet can detect if it's intersecting an entity. The logic for that is in the LaserBullet class:

```
/// <summary>
/// React to collision with an entity.
/// </summary>
/// <param name="entity"></param>
public override void OnCollideEntity(Entity entity)
{
    if (mState == ProjectileState.FreeMotion)
    {
        if (entity != this && entity != mParent)
        {
            KillEntity(entity);
        }
    }

    base.OnCollideEntity(entity);
}
```

This standardised way of creating entity interactions is used throughout the program. Interactions between different objects is one of the key challenges of game programming. We want everything to be encapsulated, and this is easy when doing internal things, like modifying our own position. What is harder is that, in order for a bullet to kill the player, either the player must know what a bullet is, or a bullet should know what a player is. Having the EntityManager be the 3rd party to mediate between the two is the best way to do this while maintaining a semblance of encapsulation.

From a base class of Entity, we proceed in an object-oriented way; we build a taxonomy of classes. From the completely abstract concept of an Entity, we slowly get more and more specific until we reach our actual in-game objects. A snippet of this taxonomy looks like this:



At the top of our tree is the Entity, then we descend until we reach each leaf node, which is an actual entity within the game, shown in blue. The key concept here is that when I want to add a new type of entity, I want to be able to make a file and describe only the behaviour that is specific to that entity. For example, when I made the “Trundle” enemy, I just described how it decides to move around the level. I don’t need to describe how to move, that is covered in MovingEntity. I don’t need to describe how to jump, that is described in PlatformingEntity. This makes adding new features very easy. It also makes solving bugs very easy; if there were a bug in how the jumping mechanics work, I only have to change PlatformingEntity, and that fix will propagate to all the child classes. On the other end of this double-edged sword is the fact that a bug in PlatformingEntity can propagate to all the child classes, thus I needed to be very careful when modifying the important base classes.

Proceeding down the tree, a moving entity is one that participates within the collision system. It can collide with other objects and the tilemap itself. Then, a platforming entity is a moving entity that has gravity, can jump, and move left and right. This became an important class and there are some things, such as projectile entities, that ended up inheriting from this class, even though one might not consider a bomb to be a platforming entity. However the bomb needed to obey the laws of gravity, and this was the easiest way to do that. In an ideal world, I might have added one more level in-between MovingEntity and PlatformingEntity.

Collisions

The collision system is easily the most flawed part of this codebase. The system is overcomplicated, very slow, and only barely works. If there is one part of this code you should not copy, it’s this.

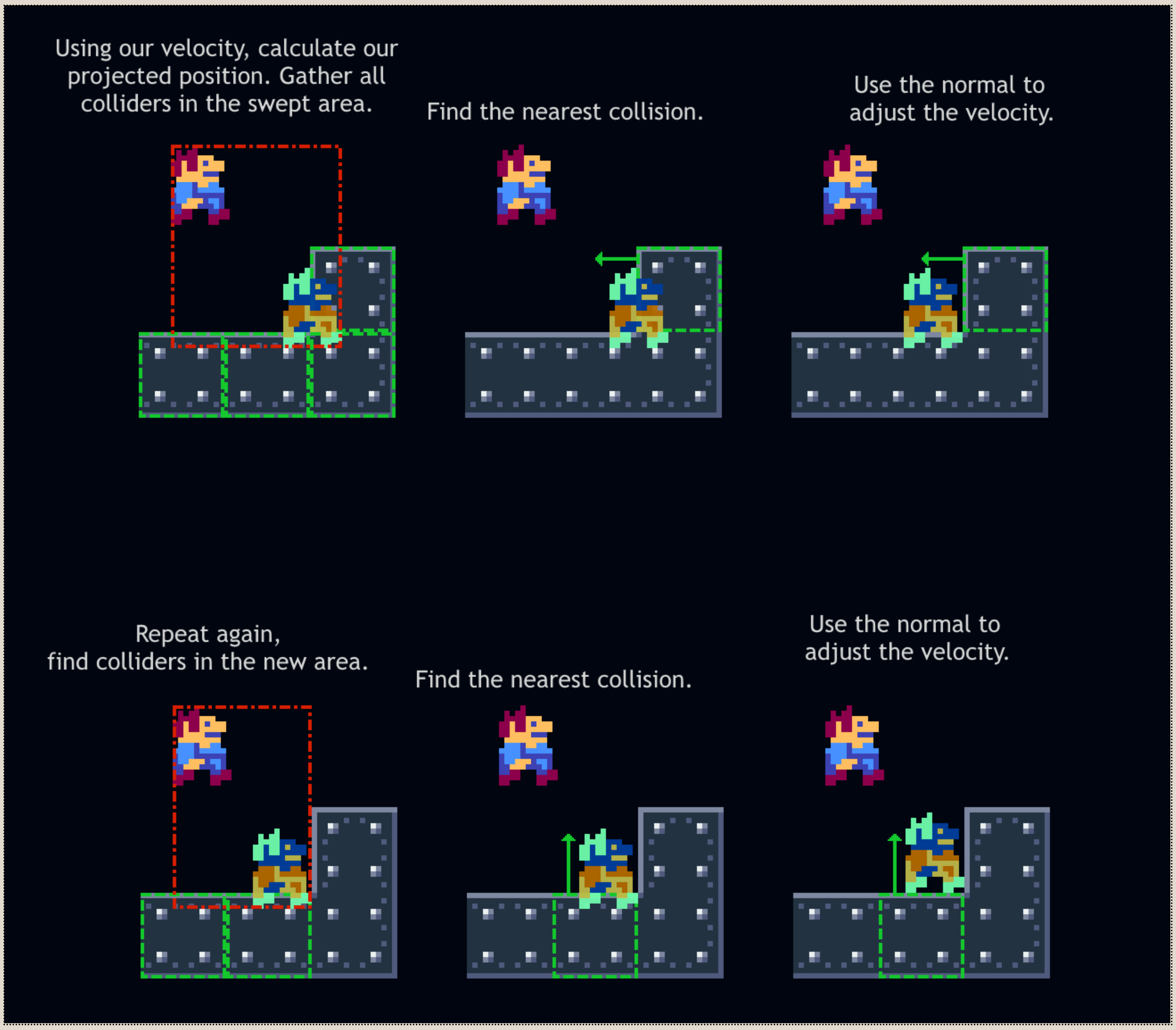
It starts out simple enough, in the file Collision.cs we have some basic collision algorithms for intersecting rectangles and rays. These are standard algorithms so I won’t be going over them. There are some details worth considering, such as if we want to consider the boundary of a rectangle to be part of the rectangle? And if we want a ray inside a rectangle to be considered as intersecting it? I am not sure what the correct answer to these questions are, I simply changed my answers until the bugs went away by trial and error. It seems the correct answer is to consider the top and left edges as part of the rectangle but not the bottom and right edges, and that internal rays do not intersect the rectangle. Is this what all games do? Did this just work by coincidence? I don't know!

For an entity to be part of the collision system, it must register itself every frame to the EntityManager. It does this with a ColliderSubmission, which represents a thing that can be collided with. Once every entity has submitted their entry to the pool, we move on to solving the collisions. This is done in MovingEntity::OrderedUpdate(), which calls MovingEntity::UpdateCollision(). You will probably notice a few hacks in here, but let's look at the algorithm without those:

```
private void UpdateCollision(GameTime gameTime)
{
    // List of all collisions that actually happened. A collision can be detected but
    // never actually happen. E.g. if we were going to collide with a wall, but the ground
    // is in the way.
    List<EntityCollision> collisionList = new List<EntityCollision>();

    EntityCollision currentCollision = EntityManager.I.GetNextCollision(gameTime, this);

    while (currentCollision != null)
    {
        EntityCollision entityCollision = currentCollision;
        CollisionResults collisionResults = entityCollision.GetResult();

        Vector2 pushVec = collisionResults.normal * new Vector2(Math.Abs(mVelocity.X),
Math.Abs(mVelocity.Y)) * (1.0f - collisionResults.t.Value) * 1.014f;
        mVelocity += pushVec;

        collisionList.Add(entityCollision);

        currentCollision = EntityManager.I.GetNextCollision(gameTime, this);
    }

    ApplyVelocity(gameTime);

    foreach (EntityCollision entityCollision in collisionList)
    {
        entityCollision.PostCollisionReact(this);
    }
}
```

So the basic outline is this, we find what we will be colliding with from the pool of all colliders. This can be calculated using our velocity and collider bounds; we sweep the area that we will travel and gather all colliders in that area. The comment explains that this is just collecting all things within our path, but it is possible that something in our path is not collided with. Imagine two walls next to each other, if I went at high speed towards them, both walls would be in the swept area. But, we would be stopped by the first wall, and never actually collide with the second wall. Getting the "next" collision means taking all these potential collisions and finding the one that is closest to us, since this is the one we would collide with first.

Once we have a collision in our path, we need to adjust our velocity so that we do not end up clipping into it. For example the ground needs to stop our y-velocity. Some collision systems opt instead to move the character then try and edit the position so it is outside of all colliders, this system tries to prevent the characters from intersecting in the first place. We adjust the velocity using the normal from our collision. We then find the next collision and repeat, we keep going until we find no collisions. At that point we can travel to our projected position.


Using our velocity, calculate our projected position. Gather all colliders in the swept area.
Find the nearest collision.
Use the normal to adjust the velocity.
Repeat again, find colliders in the new area.
Find the nearest collision.
Use the normal to adjust the velocity.

Step by step guide for collisions.

In this diagram the Arnold in inverted colors shows the projected position.

You will notice a strange factor of 1.014 added to the collisions. This is our first hack, which I call the **bodge factor**. The problem here is that even though this algorithm is mathematically sound, there are some practical hurdles that make this not work. Namely that computers don't store numbers to infinite precision. Instead, computers only store approximations of decimal numbers. This is known as floating point rounding[4]. This means every calculation we do will have rounding errors. These errors can make Arnold simply phase into a collider.

The idea with the bodge factor is that we want to push Arnold a bit further than we have to, this means Arnold has a bit of clearance between the colliders which will be greater than any rounding error. We

[4] A number like 1/3 can only be written with infinite decimals, but computers would only store a few decimals before truncating the number. Well technically, they only store a few binary digits.
https://learn.microsoft.com/en-us/cpp/build/why-floating-point-numbers-may-lose-precision?view=msvc-170

have to be careful with how big the number is, if we make it too big we could see Arnold bouncing around as he gets pushed way too far.

This hacky fix however introduces a number of new problems. The first being that since we push further than mathematically necessary, we can get stuck in an infinite loop bouncing between two colliders. It is rare, but we can account for this case by simply breaking out of the loop:

```
if (collisionList.Count > COLLISION_MAX_COUNT)
{
    // Fail-safe, don't move
    mVelocity = Vector2.Zero;

    //Clear list of bogus
    collisionList.Clear();
    break;
}
```

The next problem is that for very small velocities, multiplying a small velocity by 1.014f has some pretty significant rounding errors. To fix this, we simply ignore small movements. This has no noticeable visual effect.

```
if (mVelocity.LengthSquared() < COLLISION_MIN_VELOCITY * COLLISION_MIN_VELOCITY)
{
    mVelocity = Vector2.Zero;
}
```

Let’s now take a look at how these collision submissions are structured. They can take various forms, but all inherit from “ColliderSubmission”:

- RectangleColliderSubmission
    - The most simple submission, a static rectangle.
- EntityColliderSubmission
    - An entity submits themselves as a collider.
- PlatformColliderSubmission
    - Platform submits itself for collision.
- ReflectedTileSubmission
    - A special collider submission is how the “reflected Arnold” mechanic is coded in the mirror world. This class demonstrates the power of inheritance, it is easy to program a special mechanic by adding a new file, while leaving others untouched.

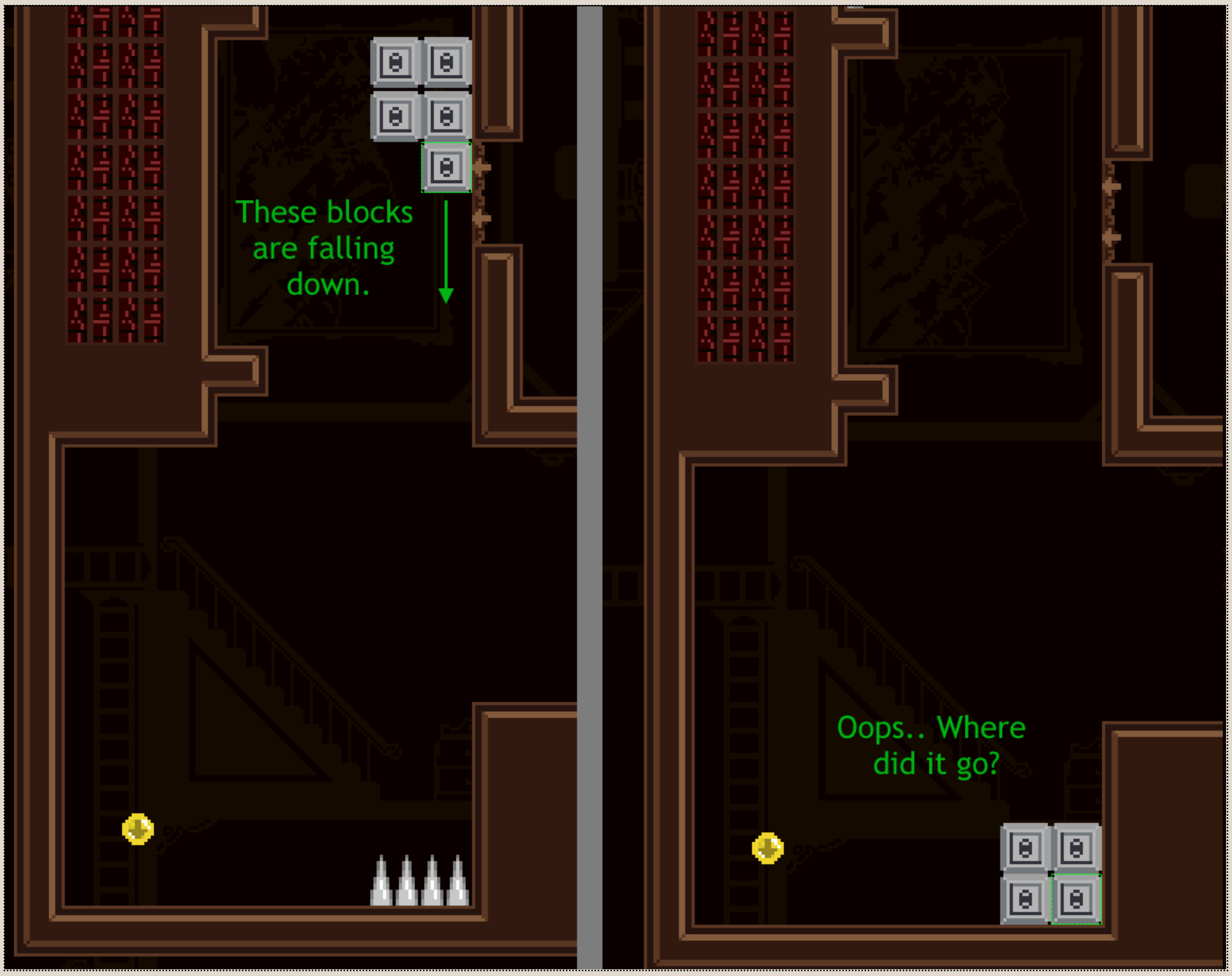
So what is the responsibility of a “ColliderSubmission”? Simply, it is a thing we can query to see if a given MovingEntity has collided with it, and if so it must generate an “EntityCollision” instance. It is important to note that an EntityColliderSubmission must store a reference to the entity itself; it can’t just store a simple rectangle. This is because in the process of resolving collisions, entities will move. Thus, to know if an entity has collided with an EntityColliderSubmission, we must know that entity’s current position, not the position at the start of the resolving process.

Querying the submission returns an EntityCollision, which stores the information about the collision, such as the normal, and time of collision. The time is used to determine which collision is “closest” in the above described algorithm. These classes also have custom behaviour for the type of collision described in the PostCollisionReact() function.

In summary, the collision system works like this:

1. Collision submissions are sent to the EntityManager, which pools them all together.
2. For every moving entity we run the collision algorithm:
    a. Project where the velocity will take us. Gather "ColliderSubmission" instances swept area.
    b. Find which of those submissions we would collide with, by querying "GetEntityCollision"
    c. Find which of those collisions has the lowest "time" variable.
    d. Using the normal, adjust the velocity.
    e. Repeat until we aren't colliding with anything, or we reach the maximum number of collisions.
3. For every collision, we call PostCollisionReact for special behaviour.

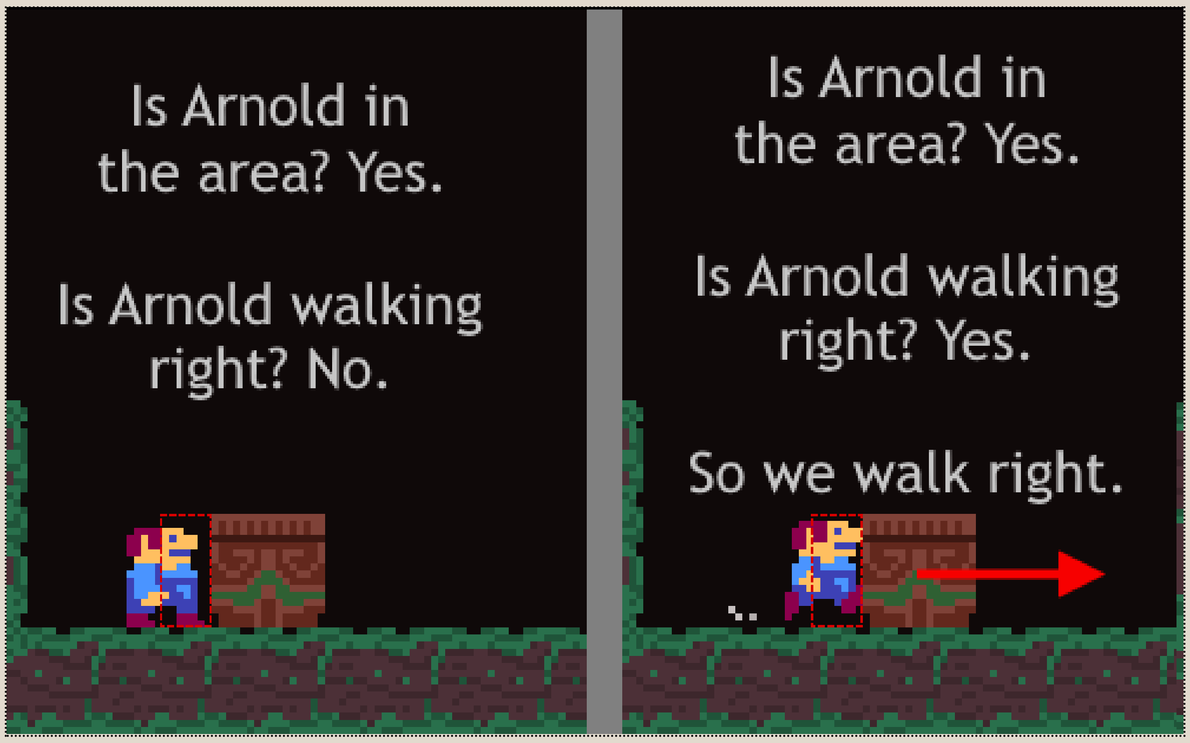
So why is this so bad? It seems perfectly cromulent on the surface. The first problem is that it doesn't always work. The "bodge" factor introduces the possibility that objects get pushed into other objects when they are stacked. As far as I am aware this bug can only been seen at one point in the game: the end of 2-4:



There are 5 at first but the bottom block clips into the rest. This is because the bodge factor will push the bottom block away from the floor too far and into the other blocks. So why not use a lower factor? Well then we run into rounding errors again... So throughout the development of this game, I had to keep bumping this factor up and down until all the bugs were gone. Or at least, find the value which had the lowest amount of bugs. This value of 1.014 was found to only have this bug, so that is acceptable.

It also isn't a very flexible system, only rectangles are supported, and every actor acts as an immovable object when resolving collisions. That is to say, that if Arnold collided with a block at a huge speed, the block would not move at all, nor would Arnold move if a block smashed into him at high speed. So I had to make several work-arounds. E.g. if a block hits Arnold while moving fast I didn't want it to stop dead as that would look unnatural, so it just kills him instead.

You may then wonder how Arnold is able to push blocks in the "Lost Kingdom" world. Well actually Arnold is not pushing the blocks. Instead those blocks are actually walking! Under the hood they are actually platforming entities like Arnold. So they have the capability to walk around and jump on their own. The logic the blocks use is to check if Arnold is to the side and walking towards the blocks, if so start walking too.


Is Arnold in the area? Yes.
Is Arnold walking right? No.
Is Arnold in the area? Yes.
Is Arnold walking right? Yes.
So we walk right.

I like to imagine plant pots have little legs we can't see. They fear Arnold so they always run away from him.

This is also what makes the clones of Arnold work relatively well. Because all the inputs are synced up between the clones, they don't need to push each other around. For example, if I jump my clone will also jump. I don't need to "push" him up because he is already going up on his own.

Still though, these weird hacks can only get you so far. One area I could never get working is having moving platforms go up and down. All the moving platforms do is move horizontally. They can achieve this by using the GetExtraVelocity() function, but this doesn't work with up and down.

As you can see, this collision system is the magic combination of being overly complicated, held together with hacks, not very capable, and not very performant, all with a healthy sprinkle of rare bugs. But these types of things are inevitable with game development. The system was just good enough to avoid a major refactor and make it into the final game.

Chapter 4: Graphics

Now that we understand the general outline of how our gameplay elements are set up, let's turn to how things are drawn in the scene.

MonoGame provides a very nice abstraction for drawing many 2D textures, known as the SpriteBatch class. Put simply, we can throw textures into the sprite batcher, and it will draw these to the render target[5]. As a very simple example:

```
SpriteBatch sb = new SpriteBatch();
sb.Begin();

sb.Draw(backgroundTexture, new Rectangle(0, 0, 800, 480), Color.White);
sb.Draw(playerTexture, new Vector2(20.0f, 20.0f), Color.White);

sb.End();
```

This acts as our endpoint that we interface with. We don't need to mess around with lower level graphics APIs like OpenGL, Vulkan, or DirectX. Even though going lower level could give us some performance gains, it simply isn't necessary for a game like this. In fact, as we will see soon, there are some pretty big ways in which the game is unoptimised.

The sprite batch is capable of all the basic things you would need for a 2D game:

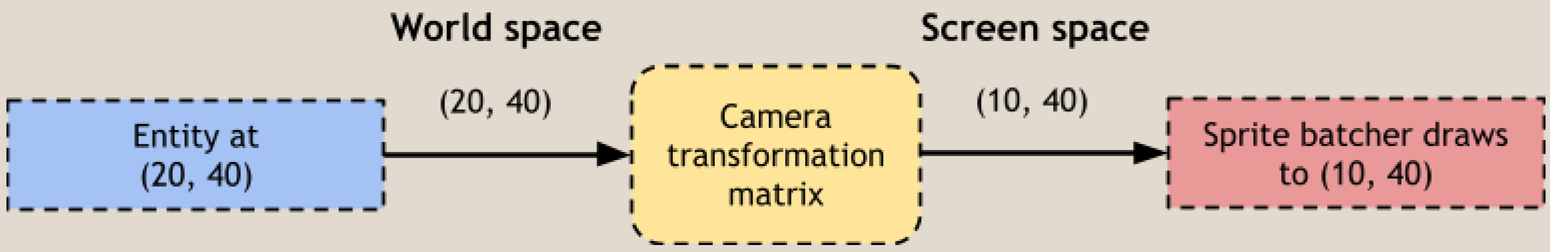
- Sprite batching(it's in the name!) to minimise draw calls.
- Rotating and scaling sprites.
- Mirroring sprites.
- Drawing sub-parts of textures.
- Can use a transform matrix.

Camera

The last point is particularly important for implementing a 2D game camera. A transformation matrix is just a fancy way of describing a particular transformation. So what is a transformation? Here is the idea: when we draw a texture to the screen, the position will be interpreted in **screen space**. That means a sprite drawn at (20, 40) will be drawn 20 pixels to the left and 40 pixels down. This is fine if you don't want a moving camera, but what if we wanted to move our viewpoint 10 pixels to the right? In that case the sprite at (20, 40) should be drawn at (10, 40). So the position of our entities in our world doesn't correspond exactly to where they should be drawn on the screen. The position of the entities is known as the **world space**.

To take a particular example from Arid Arnold, let's say I want the screen to shake. One way to do that would be to move all the sprites themselves. We could make them move up and down by shaking their actual positions, but this gets complicated because I wouldn't want that momentum to transfer to the character, and I don't want to deal with the ensuing collision bugs from moving everything all at once. In fact, moving all the entities to make them shake would be a very bad way of doing this. All I really want is their position on the screen to change, not their world space positions. We can use the transformation matrix to do that.

[5] Think of the render target as a blank canvas the game is drawing to.
https://docs.monogame.net/articles/getting_to_know/whatis/graphics/WhatIs_Render_Target.html


World space
Screen space
Entity at (20, 40)
(20, 40)
Camera transformation matrix
(10, 40)
Sprite batcher draws to (10, 40)

This lets us shake all the entities without having to actually change their positions. This answers the question "what is a matrix actually transforming?", it is transforming coordinates from **world space** to **screen space**. World space is where they are in the game, and screen space is where they are on your screen.

Let's look into the Camera.cs file to see how this works in practice (reminder to look at the Arid Arnold version and not the versions that are in the "Arcade" folder).

The first thing you will see is the CameraSpec struct. This just contains all the information needed to specify a camera's state. We have three basic parameters, position, rotation, and zoom level.

```
/// <summary>
/// Struct to specify the camera
/// </summary>
struct CameraSpec
{
    public CameraSpec()
    {
        mPosition = Vector2.Zero;
        mRotation = 0.0f;
        mZoom = 1.0f;
    }

    public Vector2 mPosition;
    public float mRotation;
    public float mZoom;
}
```

Next we have the SpriteBatchOptions struct. This just bundles all the options we can pass in at the start of a sprite batch. For more details on those, refer to the MonoGame documentation[6].

Now onto the Camera class itself. This is composed of a current spec, which stores where the camera is right now, and a current CameraMovement and a queue of CameraMovement classes. These specify which movements the camera is currently executing. For example a CameraShake is a possible movement the camera could take on. We can also queue several movements at once. So instead of needing to manipulate the camera's spec directly over several frames, we can just push on a "CameraShake" movement and forget about it.

```
screenCam.DoMovement(new CameraShake(10.0f, 2.0f, 99.0f));
```

[6] Documentation for SpriteBatch - https://docs.monogame.net/api/Microsoft.Xna.Framework.Graphics.SpriteBatch.html

This is all handled in the update function:

```
public void Update(GameTime gameTime)
{
    // See if there is a new camera movement.
    CheckQueue();

    // Perform camera movement if needed.
    if (mCurrentCameraMovement is not null)
    {
        mCurrentCameraMovement.Update(gameTime);
        mCurrentSpec = mCurrentCameraMovement.GetCurrentSpec();
    }
}
```

So how do I use this to construct our transformation matrix? Under the hood, a camera matrix is a 4-by-4 grid of numbers. Manipulating these numbers directly can be very confusing and involves some heavy maths, but luckily we can avoid this by simply combining several basic transformations. We never have to worry about what is going on mathematically. To combine two matrices we can just multiply them:

```
Matrix scaleThenRotate = Matrix.CreateRotationZ(mCurrentSpec.mRotation) *
                    Matrix.CreateScale(new Vector3(mCurrentSpec.mZoom, mCurrentSpec.mZoom, 1));
```

So here we combine a scale transformation(i.e. scaling size up or down) with a rotation. Note that the order in which these apply is right-to-left, and not left-to-right. This is due to mathematical reasons that I won't go into. Just know that the order matters, and that this example is first going to scale things up, then it will perform the rotation. The entire stack of transformations looks like this:

```
Matrix CalculateMatrix(Vector2 viewPortSize)
{
    Vector3 centrePoint3 = new Vector3(viewPortSize / 2.0f, 0.0f);

    return Matrix.CreateTranslation(-(int)mCurrentSpec.mPosition.X, -(int)mCurrentSpec.mPosition.Y,
0) *
            Matrix.CreateTranslation(-centrePoint3) *
            Matrix.CreateRotationZ(mCurrentSpec.mRotation) *
            Matrix.CreateScale(new Vector3(mCurrentSpec.mZoom, mCurrentSpec.mZoom, 1)) *
            Matrix.CreateTranslation(centrePoint3);
}
```

Again, reading from left-to-right(or bottom-to-top in this case) we are defining how we transform from world space to screen space:

1. First move everything so that (0,0) lands into the middle of the screen.
2. Now scale by the zoom level.
3. Rotate everything by our camera's rotation.
4. Undo the translation from step 1.
5. Move the camera by minus the position.

The reason we move the camera by minus the position is because when the camera moves left, the things on the screen should move right. Pull out your phone and open the camera app, find something to focus on and aim your phone so that it is in the middle of the screen. Hold your hand steady without rotating your arm, and walk left. The thing you were focusing on will move right on your phone screen.

The other question you might have is: why do we bother with step 1 if we are just going to undo it in step 4? The reason is that we want the things to scale and rotate about the middle of the screen. If I didn't have these two steps, things would rotate about the corner of the screen instead. The CreateRotationZ function will always rotate about (0, 0), so we need to temporarily shift things to get it to rotate around the centre of the screen.

Now, using this, we can start the sprite batch. You can think of this like asking the camera to start recording. When we call EndSpriteBatch() we stop recording, and the render target is like the piece of film we are recording to.

```
public void StartSpriteBatch(DrawInfo info, Vector2 viewPortSize)
{
    info.spriteBatch.Begin(mSpriteBatchOptions.mSortMode,
                            mSpriteBatchOptions.mBlend,
                            mSpriteBatchOptions.mSamplerState,
                            mSpriteBatchOptions.mDepthStencilState,
                            mSpriteBatchOptions.mRasterizerState,
                            null,
                            CalculateMatrix(viewPortSize));
}
```

Rendering a frame

Now that we have the camera setup, let's look at the flow of graphics. We will look at the game screen since that is the main part of the game. Some games use multiple sprite batches but in Arid Arnold we just have one and throw everything in it. We will be starting and stopping it once for every camera in order to change the render target.

There are a few things that every draw function will need, like the current sprite batch, graphics device, and delta time. I bundled these into a struct that is easy to pass around, called DrawInfo:

```
public struct DrawInfo
{
    public GameTime gameTime;
    public SpriteBatch spriteBatch;
    public GraphicsDeviceManager graphics;
    public GraphicsDevice device;
}
```

This is a handy tip to avoid long function signatures. You can just bundle common parameters together into a struct.

In our Main.cs we fill in these fields, and from there they are only read, not written to.

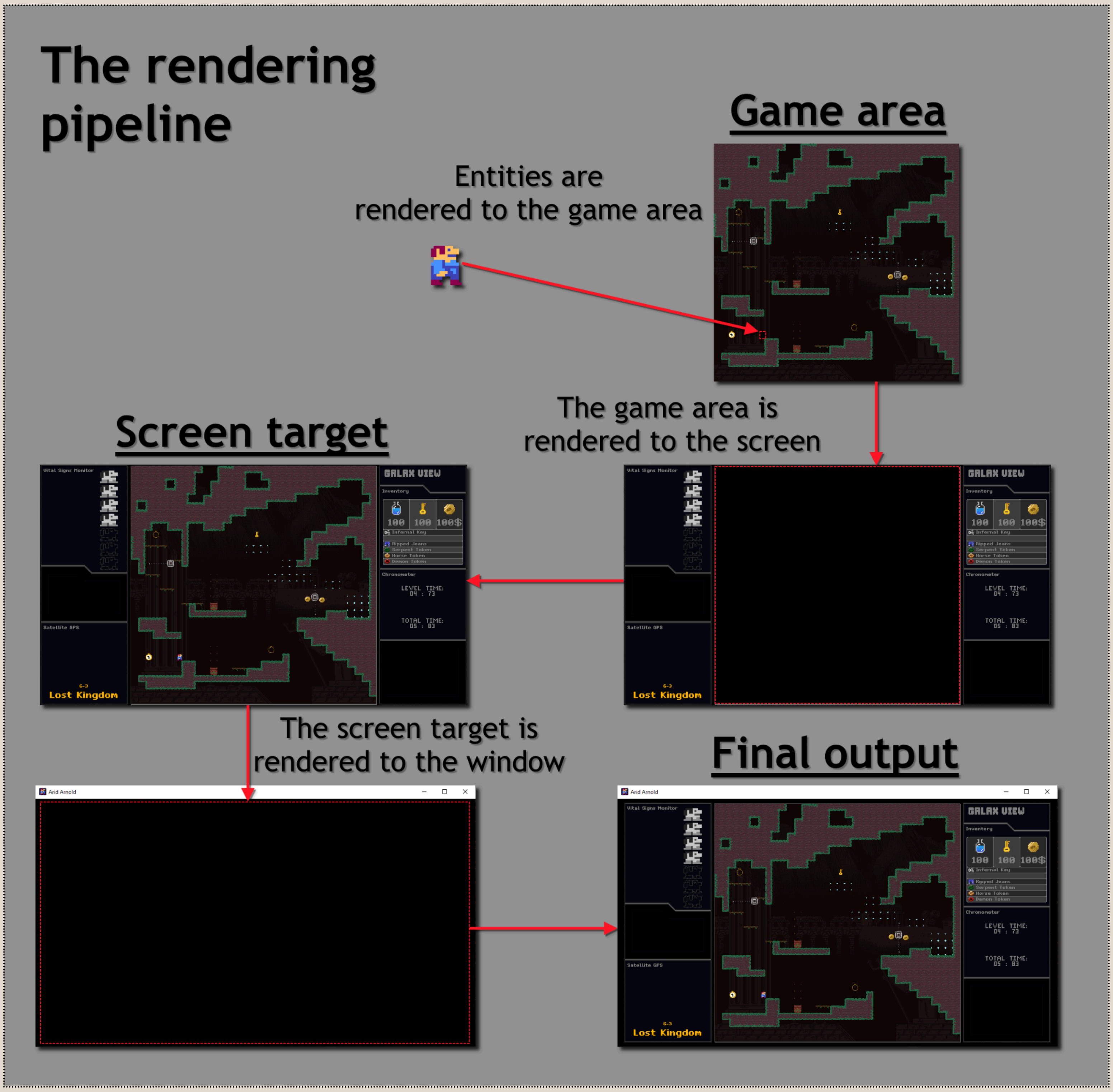
```
protected override void Draw(GameTime gameTime)
{
    DrawInfo frameInfo;

    frameInfo.graphics = mGraphicsManager;
    frameInfo.spriteBatch = mMainSpriteBatch;
    frameInfo.gameTime = gameTime;
    frameInfo.device = GraphicsDevice;

    // Send it down to the screen...
}
```

This draw info is then sent down to the screens. As a reminder, the screens draw to their own render target, which is then later drawn into the window, and scaled to fit. This is done in DrawToRenderTarget() for each screen. This is what that function looks like for the GameScreen class:

```
public override RenderTarget2D DrawToRenderTarget(DrawInfo info)
{
    //Get game rendered as a texture
    RenderGameAreaToTarget(info);

    //Draw out the game area
    StartScreenSpriteBatch(info);

    if (!mFadeInFx.Finished())
    {
        mFadeInFx.Draw(info);
    }
    mMainUI.Draw(info);
    DrawGameArea(info);

    mPauseMenu.Draw(info);

    EndScreenSpriteBatch(info);

    return mScreenTarget;
}
```

First we render the game area to its own separate target in RenderGameAreaToTarget(). This is the box in which the main game happens. This function gives us a texture which can then be drawn to the screen’s render target. After that we draw the UI, and then composite the game area on top of that. This finishes our screen’s render target, which will then be sent to Main.cs to be drawn to the actual screen.

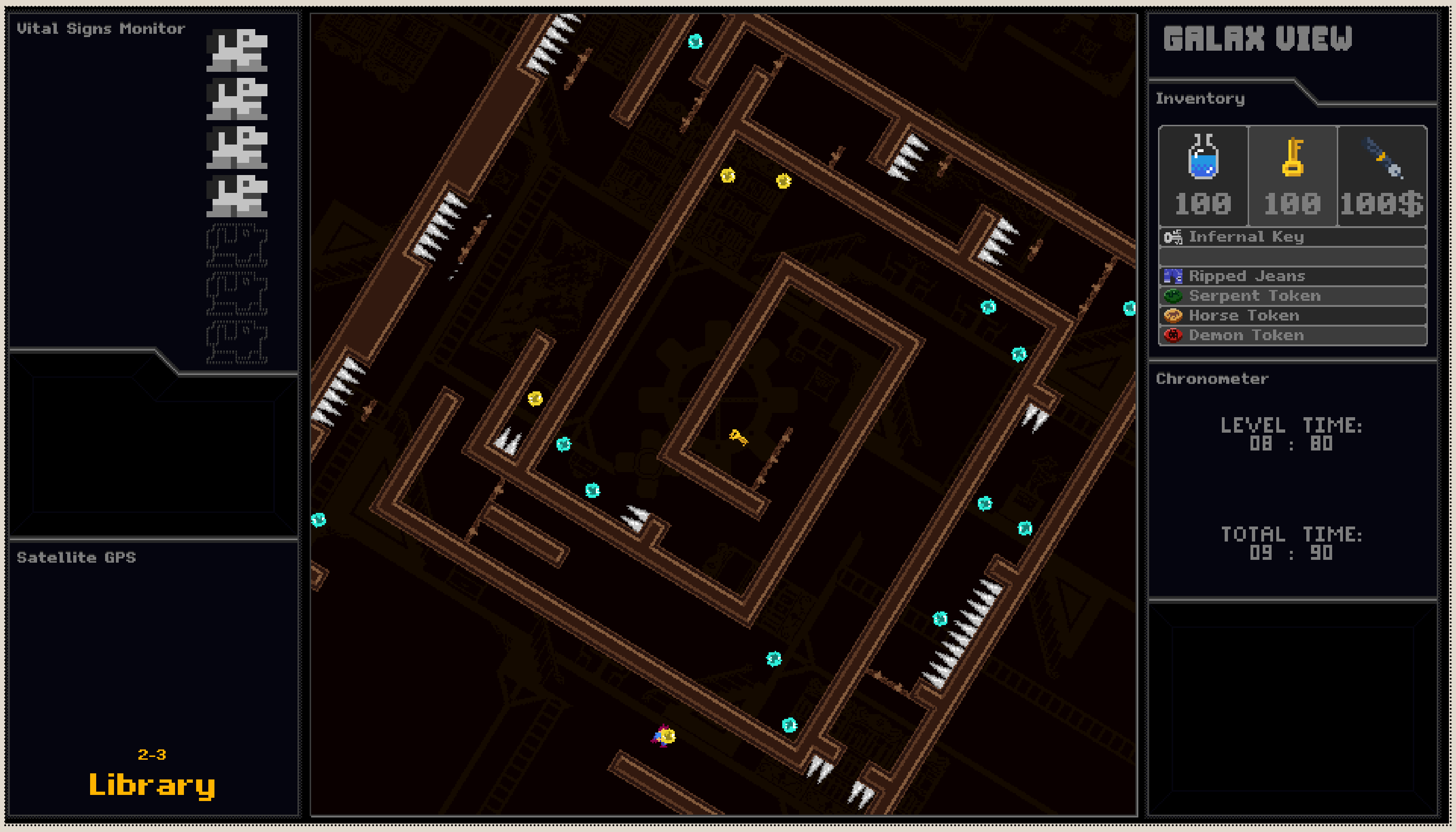
In summary the pipeline looks like this:


The rendering pipeline
Game area
Entities are rendered to the game area
The game area is rendered to the screen
Screen target
The screen target is rendered to the window
Final output
Lost Kingdom
Arid Arnold

Note that for each new render target we have a separate camera. This means they all have their own coordinate spaces. For example if I draw an entity at (0,0) it won't appear in the top left of the window, it will appear in the top left of the game area render target.

This distinction is also important because it affects how we want things to move. If I want to rotate the game's camera I can push that CameraMovement to that camera only, and only the Game Area will rotate, the UI will not rotate with it.

For example, consider in the Library world when the camera is rotating[7] after touching an orb:



In some cases, like when I want to have a big impact, I want to apply a movement to the whole game, UI included. When the demon boss dies I apply a camera shake to the global camera, meaning that everything shakes all at once, including the UI. This creates a bigger impact than just shaking the game area.

Animations

Looking through the data of the game, you might notice something weird: all the textures are separate. There are no sprite sheets. The reason for this is purely convenience. An image file has information about height and width encoded within and so it makes it easy to pass around and load. On the other hand, a sprite sheet requires a separate file external to it which defines the position and sizes of each sprite within. You then have to load both files and match up the frames for them to be passed around. This also requires a special tool to easily generate the files, lest I spend hours creating them by hand. Using separate images for each animation frame is simply more convenient.

The disadvantage of this is performance. I mentioned earlier that I was using the sprite batcher, and that this can “batch” several draw calls into one. Well it only works if we draw several of the same texture in a row. Since every frame is its own texture, we effectively get no batching! So what did I do about this problem? Simply not care and move on. Performance was never a problem for me, so I never had to look into using sprite sheets.

The Animator class is fairly straightforward. There are several frames it goes through, and it can either play once or repeat. There is a member called mPlayHead which tracks how far into the animation we are. Based on that, we can get the current texture.

[7] Although it might seem like the whole level is rotating, it is actually staying still while just the camera is rotating. We then change Arnold’s gravity direction so it looks like he is falling “down”.

I can define these animations in external files. These are called .max files, but are actually just XML files under the hood:

```
<animation type="repeat">
    <texture time="0.1">BG\WW7\Missile1</texture>
    <texture time="0.1">BG\WW7\Missile2</texture>
    <texture time="0.1">BG\WW7\Missile3</texture>
</animation>
```

In general, it is good to move data out to external files, rather than having it stored within the code. This decreases code clutter, and makes things easier to find. It also helps for adding new content to the game, because this becomes a matter of adding files rather than adding code. Eventually, the benefits start to compound, since pieces of data can easily reference each other. For example, an idle animation cycle is defined by composing various animations:

```
<animation variation="45.0">
    <main>NPC/BoilerMan/SpannerWait.max</main>
    <alt>NPC/BoilerMan/TightenBolt.max</alt>
    <alt>NPC/BoilerMan/Rest.max</alt>
</animation>
```

Here we have a default idle animation, and a couple of variations we can cycle to. We are normally in the SpannerWait animation, but at the end of that we have a 45% chance to go to either TightenBolt or Rest. This small amount of random chance makes the NPC idle animations a bit less repetitive. The fact that I have extracted the animation data means other pieces of data can then be more easily defined. For comparison, here is how this would be defined in code:

```
IdleAnimator idleAnimation = new IdleAnimator(
    new Animator(Animator.PlayType.OneShot, ("NPC/BoilerMan/Default", 0.6f)),
    45.0f,
    new Animator(Animator.PlayType.OneShot, ("NPC/BoilerMan/Default", 0.2f),
                                            ("NPC/BoilerMan/Tighten1", 0.2f),
                                            ("NPC/BoilerMan/Default", 0.2f),
                                            ("NPC/BoilerMan/Tighten1", 0.2f)),
    new Animator(Animator.PlayType.OneShot, ("NPC/BoilerMan/Rest1", 0.8f)));
```

Once we have animations defined entirely in files, we can compose several of them to define NPCs. If you think about it, the only differences between NPCs are visual. We can then define them entirely in data as a collection of animation files. This made the process of adding new NPCs dramatically easier, since I could just copy an existing NPC's data and modify it, without needing to touch any code. I hope this demonstrates the benefits of extracting data from your code. If you do it enough, your game development work will be editing data rather than editing code.

Geometry

One thing lacking from the sprite batcher is the ability to draw geometric shapes, it can only draw 2d textures. For example, there is no in-built function to draw a line or a triangle. This limitation can be somewhat surmounted by clever use of the sprite batcher. First we create a sprite which is a single white pixel. This is called the dummy texture.

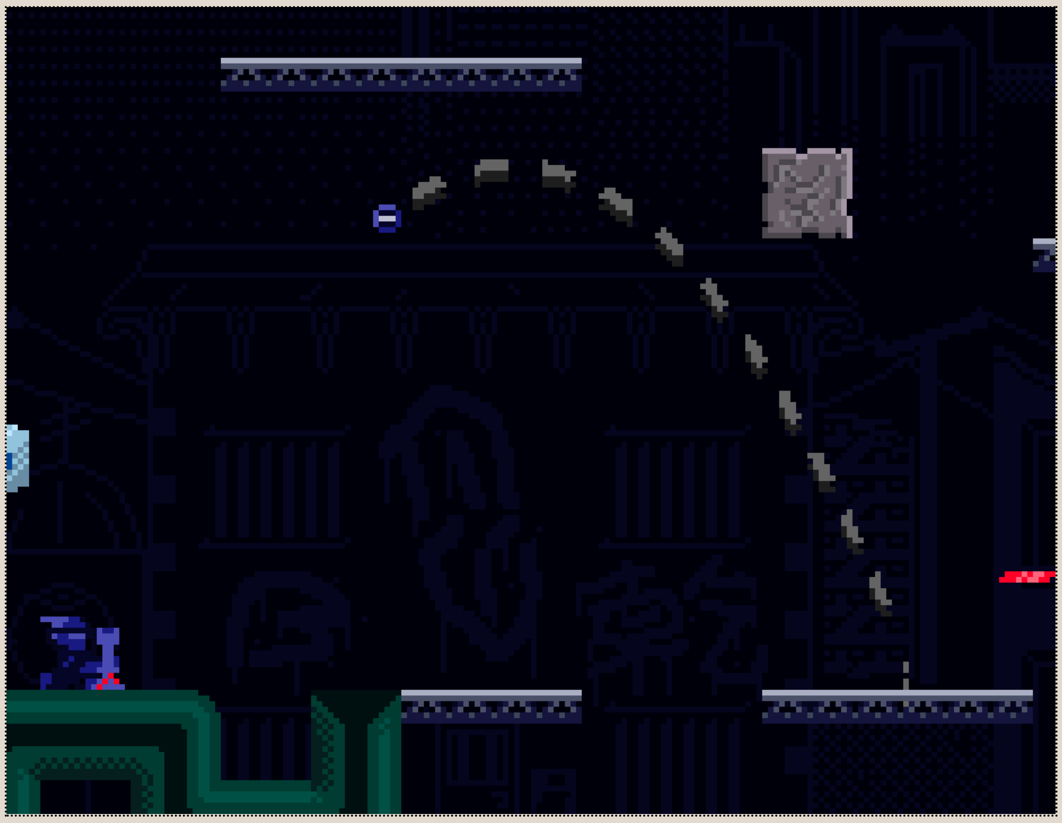
The sprite batcher can then scale and color this dummy texture to draw any rectangle of any color:

```
public static void DrawRect(DrawInfo info, Rectangle rect, Color col)
{
    info.spriteBatch.Draw(Main.GetDummyTexture(), rect, col);
}
```

In fact, by stretching and rotating a rectangle we can draw any line:

```
public static void DrawLine(DrawInfo info, Vector2 point, float length, float angle, Color color,
float thickness = 1.0f, DrawLayer depth = DrawLayer.Default)
{
    var origin = new Vector2(0f, 0.5f);
    var scale = new Vector2(length, thickness);
    info.spriteBatch.Draw(Main.GetDummyTexture(), point, null, color, angle, origin, scale,
SpriteEffects.None, GetDepth(depth));
}
```

Using these two functions, we can generate a number of geometric shapes. For example in the "World War 7" levels I drew parabolic arcs out of several small line segments:



There are many of these drawing utilities, all stored in the MonoDraw.cs file.

Layouts & UI

When it came time to add background elements to the levels, I needed an easy way to position things such as animated sprites or images. To solve this, I did the simplest possible solution and encoded these positions and animations in an xml file, which I called a layout file.

For example, let’s look at part of the steam plant’s background layout:

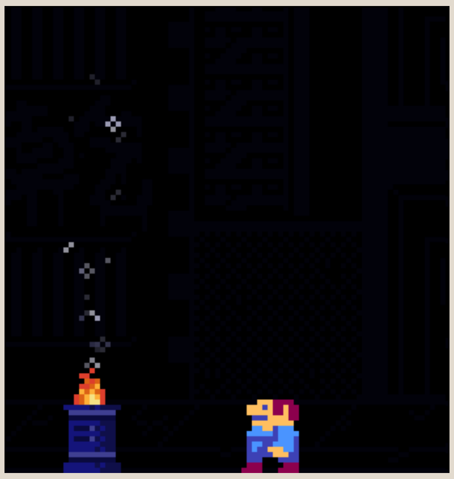
```xml
<?xml version="1.0" encoding="UTF-8" ?>
<root>
    <element type="texture">
        <x>0.0</x>
        <y>0.0</y>
        <depth>Background</depth>
        <texture>BG/SteamPlant/BG1</texture>
    </element>
    <element type="IdleAnim">
        <x>200.0</x>
        <y>394.0</y>
        <depth>BackgroundElement</depth>
        <anim>BG/SteamPlant/DialIdle.mia</anim>
    </element>
    <!-- more animations ... -->
</root>
```

The first element is at (0,0) and is a simple texture. The next element is an animation at (200, 394). So the format here is pretty simple, we can specify types such as “texture” or “idleAnim” then pass in parameters. When reading the file, the game uses the C# reflection system. This allows us to convert a string “IdleAnim” into a class that we can instantiate. In this case, we add an “E” to the start and turn the string into “EIdleAnim”, which is our class name. The class will then read the inner xml tags to find which animation to play. This simple system lets me place elements in the background. The same system is used to define the UI layouts. Luckily for me, I chose a fixed resolution. This meant I could just specify all the button positions in absolute pixels and be done with it.

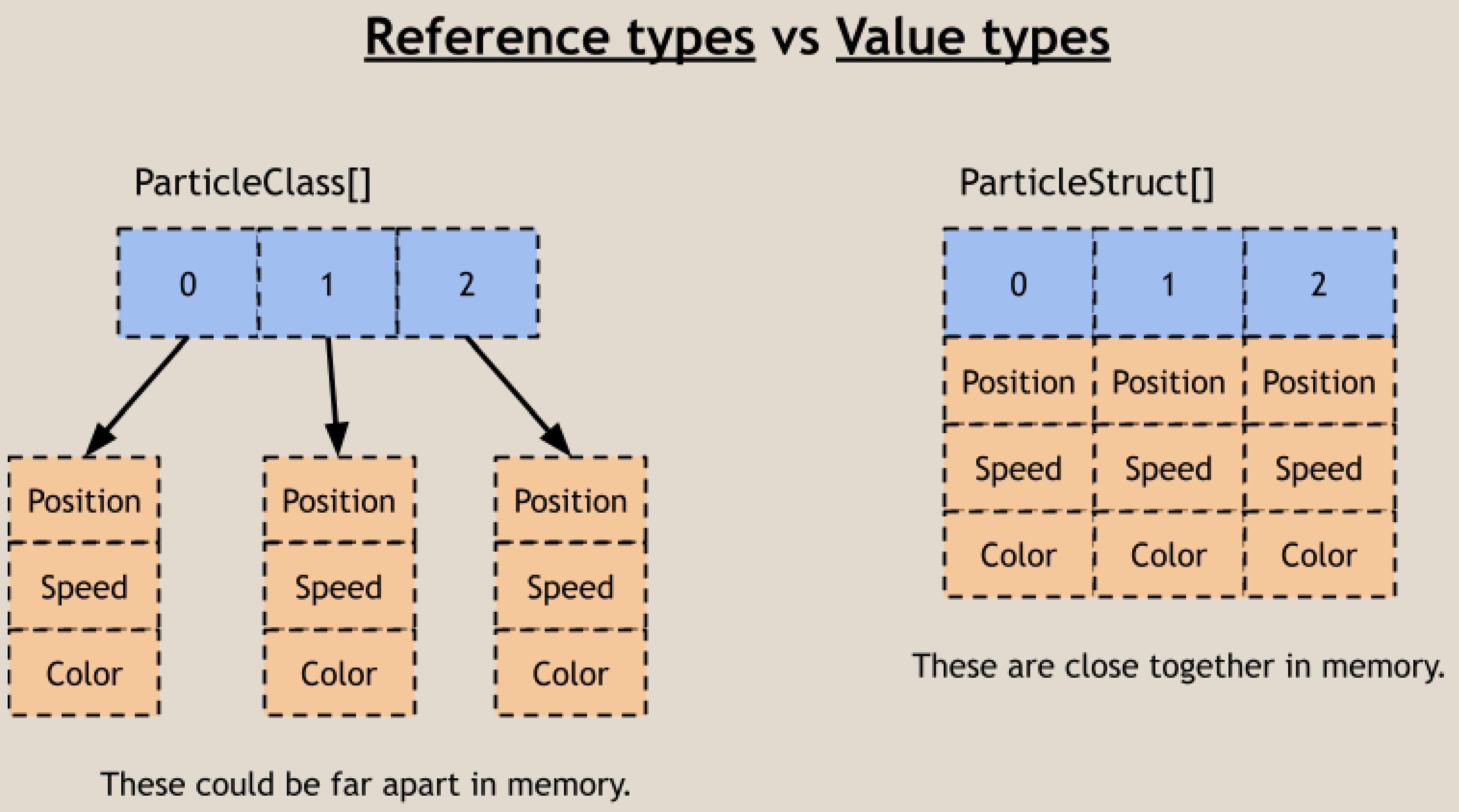
Effects & Particles

Effects in the game are things which have a visual element but do not affect gameplay. This is pretty simple to implement, we just have these classes that have Update() and Draw() methods and chuck them into FXManager. The FXManager will then update and draw them. We can then implement these two methods in various ways to draw any effect we want.

Particles are where things get a bit more interesting. We want to draw a lot of small things at once, so we need to be careful to make this more performant than the simple FX class. Consider this example where a barrel wants to spawn many smoke particles:



Each particle needs its own texture, position, color, and trajectory. The naive way to do it would be to make a SmokeParticle class which inherits from FX, and add it to the FXManager. The problem with that is that classes in C# are reference types. A reference type has 1 layer of indirection, meaning the class isn't stored directly in our variables, but rather our variables hold a pointer that is a reference to the value[8]. The problem for performance is that holding an array of such values would mean the particles would be fragmented in memory, rather than being stored contiguously. To illustrate:


Reference types vs Value types
ParticleClass[]
0
1
2
Position
Speed
Color
These could be far apart in memory.
ParticleStruct[]
These are close together in memory.

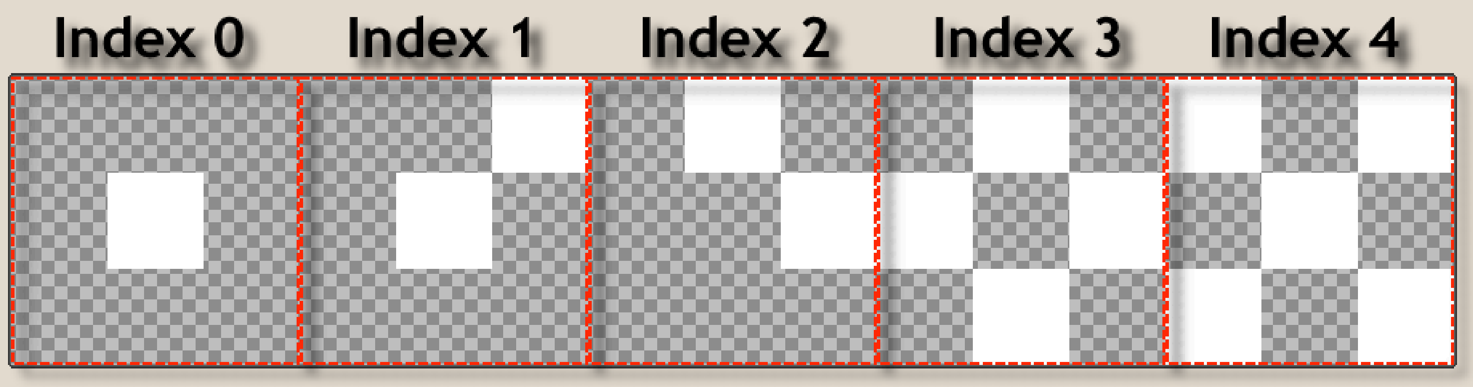
In terms of performance the class (reference type) will be worse, because now the CPU has to jump all over the place in RAM, which adds extra operations and is worse for cache efficiency[9]. It also helps that structs will have exactly the same size whereas classes can have varying sizes. The struct for the particles is described in the Particle.cs file:

```
struct Particle
{
    public Color mColor;
    public Vector2 mPosition;
    public Vector2 mVelocity;
    public byte mFlags;
    public byte mTextureIndex;
    public ushort mLifetime;
}
```

We store the color, velocity and position, which should be self-explanatory. We also have some flags in a byte so we can tag certain particles for custom behaviour. The "mTextureIndex" is used in place of a UV[10]. Each particle will be using the same texture as a sprite sheet. The index shows which sprite from that texture we want this particle to use.

[8] Reference types:https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types
[9] Adam Sitnik, Performance of value types vs reference types: https://adamsitnik.com/Value-Types-vs-Reference-Types/
[10] Wikipedia, UV Mapping: https://en.wikipedia.org/wiki/UV_mapping

For example, here is what the smoke particles texture looks like:



The last parameter is the lifetime, this value is decremented once per frame. Once it reaches zero we delete the particle.

The particles are handled in the ParticleSet class. This holds an array of these particles and handles their allocation. Again, because of the high number of particles that are spawned in and deleted on each frame, allocating them anew each time would cause problems with the garbage collector[11]. So instead I allocate a fixed array at application launch. This array is filled with particles from the beginning but that does not mean we will actually be drawing those particles, they are just dummy particles for now. Imagine it like a restaurant hiring 500 people at the beginning of their business just to avoid having to pay the cost of hiring and firing people. There aren't always 500 people working in the restaurant, but they still exist and can be put to work at any moment.

I keep track of how many particles we are actively displaying in mParticleHead. I.e. The particles we are actually using are between the index of 0 and mParticleHead. This makes adding a new particle very quick, we just have to write to the array at mParticleHead:

```
public void AddParticle(ref Particle particle)
{
    if(mParticleHead >= MAX_PARTICLES)
    {
        MonoDebug.Log("PARTICLE MAX EXCEEDED. Consider upping maximum");
        return;
    }

    mParticles[mParticleHead] = particle;
    mParticleHead++;
}
```

So here we add a particle to the end of the array, then move the array head forward by one.

Deleting is a bit more tricky. If we wanted to delete the particle at the end of the array we could just move the particle head backwards by one. However, what about if we want to delete an element in the middle of the array? Well the trick is to just reorder the array so that the element we want to delete is at the end. Be the change you want to see in the world.

[11] Garbage collection and performance: https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/performance

```
public void RemoveParticle(int index)
{
    if(index != mParticleHead-1)
    {
        // Move end to the index we are deleting to overwrite it.
        mParticles[index] = mParticles[mParticleHead - 1];
    }

    mParticleHead--;
}
```

This means we can quickly add and remove particles without ever having to allocate or deallocate anything. This is a bit cumbersome and not memory efficient, our array stores a lot of memory that isn’t strictly needed. So it is a tradeoff between speed and memory usage. Now we can draw our particles:

```
public void Draw(DrawInfo info)
{
    Rectangle baseRect = new Rectangle(Point.Zero, mBaseSize);

    for (int i = 0; i < mParticleHead; i++)
    {
        ref Particle particleRef = ref mParticles[i];

        baseRect.X = mBaseSize.X * particleRef.mTextureIndex;
        MonoDraw.DrawTexture(info, mTexture, particleRef.mPosition, baseRect, particleRef.mColor,
0.0f, Vector2.Zero, 1.0f, SpriteEffects.None, DrawLayer.Particle);
    }
}
```

There’s a couple of things to note here. First of all, the particles all draw from the same texture. Secondly, the particles are drawn on their own separate layer. This ensures that they will be drawn one after another. These two conditions are necessary for the sprite batcher to bundle all these draws into one. If we had instead allowed them to use separate textures, this could have separated the draw calls and resulted in potentially thousands of draw calls.

Having said all of this, I do think that I could have gotten away with doing the naive solution. The number of particles that are typically on screen at any given time likely never exceeds 100, so this level of optimization wasn’t really needed. It was mainly just for personal interest that I pursued this idea instead of using the FXManager.

Chapter 5: Audio

Audio is one of those areas in which abstraction is used the most. It's not uncommon to see an audio library stack 3 or 4 levels high. I once worked on a project that used FMOD running on a DirectSound compatibility layer which was implemented with OpenAL(MojoAL) running on top of SDL2 which was sending audio to the console's SDK(so that is FMOD -> DirectSound -> OpenAL -> SDL2 -> SDK). The reason for all the abstraction is that, at the lowest level, audio is messy and complicated; it requires manipulating and queueing buffers at very high speed. It is more of a data streaming problem than anything. The middle level is where we think about mixing, effects, EQ, all the things an audio engineer would worry about. At the highest level, the programmer just wants to call something like PlaySound() and have a sound play. The disparity between these 3 levels means we need a separate layer of abstraction for each. For example, something like FMOD lies on the level of the audio engineer/programmer. OpenAL lies a little below that, where it can do some effects but also requires you manually queue buffers. Then SDL2 is really at the lower level, sending those buffers to the operating system.

MonoGame uses different stacks for the various different platforms it supports, but the exact stack isn't important since I only interface with MonoGame itself, nothing below it. However there are some deep flaws with MonoGame's sound library. At the time of writing, there is no way to play looping background music other than using uncompressed .wav files. Using these files would make my small 2D pixel art game over a gigabyte in size, the majority of which being from the long music tracks.

So I had considered that I might want to use an external library called MonoSound, which is meant to fix this very issue by allowing me to stream compressed audio formats like ogg. Since I had the choice between two different audio backends, I decided I should write one more layer of abstraction to unite their APIs and allow me to quickly switch back and forth. This is what is going on in the Sound\Impl folder. There is the abstract AudioBuffer which is what we use to play sounds, and SoundImplementation which can load sound files and create these AudioBuffers. The files are mostly boilerplate that interface with the underlying audio APIs, and note that they support 3D spatial audio. A position can be set as the sound's source, and then another position can be set as the listener. MonoGame will then simulate the sounds as if you were there; a sound to the left will play in the left speaker. You may notice that the MonoSound implementation is not being compiled, that will be explained later.

Once we have that layer of abstraction done, all we need to do is swap out one line of code to change our entire audio backend. Isn't inheritance great! The declaration is here:

```
static class MonoSound
{
    // Swap this out to change implementation.
    static SoundImplementation mImpl = new MonoGameImpl();

    public static SoundImplementation Impl { get { return mImpl; } }
}
```

Now that is set up, we can build out some basic classes. First of all the BufferPlayer class provides a way to play buffers with fade-in and fade-out. From there we create GameSFX, and SpacialSFX. These are just wrapper classes that provide special information to the buffer, SpacialSFX provides the position information for the spatial audio, while GameSFX just plays in both ears equally. The SFXManager is then the endpoint that we access to play sounds easily. For example when a bomb explosion goes off:

```
mExplodeSFX?.SetPosition(mPosition);
SFXManager.I.PlaySFX(mExplodeSFX);
```

In order to easily reference all the different sound effects, I created an enum. We use attributes to attach file paths directly to the enum values.

```
enum AridArnoldSFX
{
    // Menu
    [FilePath("Sound/SFX/Menu/Confirm")]                MenuConfirm,
    [FilePath("Sound/SFX/Menu/Select")]                 MenuSelect,

    // Arnold
    [FilePath("Sound/SFX/Arnold/Jump")]                 ArnoldJump,
    [FilePath("Sound/SFX/Arnold/Walk")]                 ArnoldWalk,
    [FilePath("Sound/SFX/Arnold/Death")]                ArnoldDeath,
    [FilePath("Sound/SFX/Arnold/Land")]                 ArnoldLand,
}
```

The reason for this is so I don't risk typing the wrong file path. Imagine I accidentally typed "Sound/SFX/Arnold/Wlak", the game would only crash the moment I tried to play that sound effect. The QA process is just me playing the game, so it wouldn't be that unlikely that I typo a sound effect and accidentally create a rare crash that gets missed before release. With an enum like this I can quickly write a function to go through all values and verify the file at the path exists for all of them. From there, it is a fool-proof system since I never have to type the file paths again. Potentially I could have done the same with all the other assets but never thought to when doing the initial setup. It was only because I did the audio later that I added this safety feature.

Using this to setup a sound effect looks like this:

```
SpacialSFX laserTravel = new SpacialSFX(AridArnoldSFX.FutronLaser, mPosition, 0.5f, 0.0f, 0.1f);
laserTravel.GetBuffer().SetLoop(true);
```

This can then be sent to the SFXManager to be played whenever.

For music we have a very similar setup, a MusicTrack class which is similar to a GameSFX and a MusicManager class which is similar to SFXManager. Reading through these should be fairly self-explanatory. After all, the only difference between a sound effect and a song is the length, thus both use the same AudioBuffer interface.

Threading troubles

A few days before the initial release day of Arid Arnold I noticed that the game would sometimes randomly crash. It seemed to be referencing a null pointer[12] in the MonoSound library. I double checked my code several times, even adding many null checks before accessing any buffer, but the problem never seemed to go away. The only explanation seemed to be a threading issue. So I put a bunch of "lock{}" statements in the MonoSound implementation but the problem never quite went away. Eventually the release day came and I hadn't tracked down the bug. I either had the choice to add about 500MB in extra audio files by having them run through MonoGame uncompressed, or ship with the bug. I decided that the bug was rare enough that I kept the MonoSound implementation, even knowing there was a rare crash in there.

[12] You're dereferencing a null pointer: https://youtu.be/bLHL75H_VEM?si=o_AZ8h5_R03ZzKOd

I raised the issue within the MonoGame community to see if there was an alternate solution. It seemed bizarre to me that there should be no way to stream compressed audio in MonoGame[13]. Luckily there was a way. MonoGame does support only wav files, but did you know that wav files can contain compressed audio? This is called ADPCM audio. This encoding is actually pretty bad and the compression ratios achieved are poor when compared to something like ogg or mp3, but it's better than completely uncompressed audio. Great! So let's put that in the game, right? Well for some reason it doesn't loop properly… there is a noticeable pause at the end of every track. Well there goes that idea… until a user by the name of SquareBananas told me they had identified the source of the problem. Turns out MonoGame is not at fault here but rather this was a bug within ffmpeg. Better still, they had a custom version of ffmpeg with this bug fixed. So finally in the latest version of the game, I managed to cut out MonoSound and replace it with pure MonoGame, using the ADPCM audio to compress the files to an acceptable level. The abstraction between MonoSound and MonoGame paid off in the end; switching the implementation was just a 1 line change.

Music

All of the music in "Arid Arnold" was made in Ableton. I had originally wanted to go for a purely 8-bit sound, even going so far as to limit myself to 4 channels of audio. But it turns out making 8-bit music sound good is very difficult. Those old video game composers were very talented. So instead I decided to expand the scope to "Synth Music" instead.

The music itself isn't groundbreaking or anything but there's a few interesting notes:

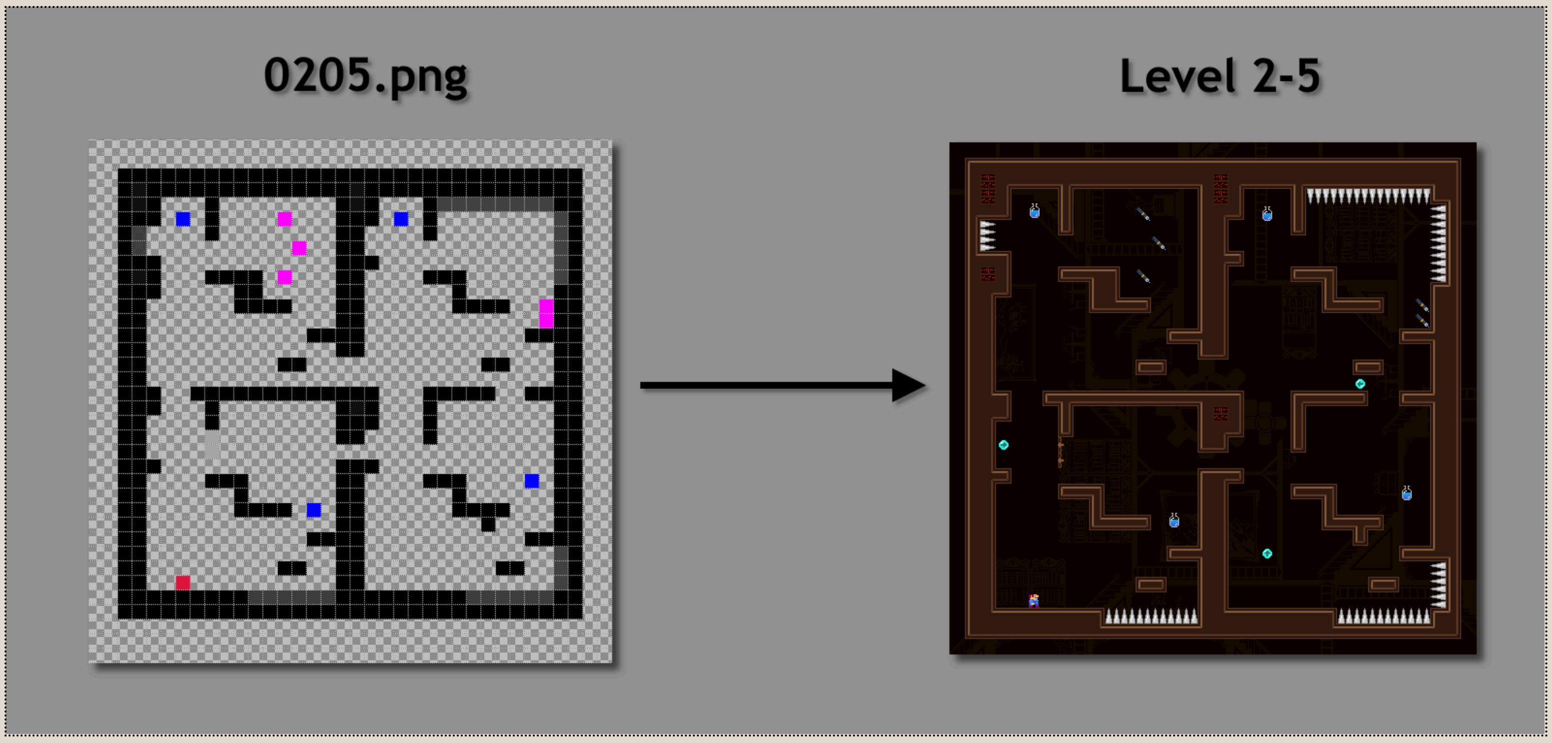
- The music in this game is very inspired by "American Football", using repetitive arpeggios and odd time signatures. I like the meditative nature of their songs, and I thought it would fit well for background music. The song for the library was inspired by their track "Stay Home".
- A lot of the songs use just intonation instead of the usual 12 tone equal temperament(12-TET). This is where we use integer mathematical ratios to define the notes within the key. For example, a fifth is defined as a 3:2 ratio from the root note. These ratios sound far more in tune than 12-TET, but the drawback is that you can't change keys. This is why it is not the standard. But I know my songs do not change keys. The music in "the lab" uses this to create a very pleasant drone effect.
- I had struggled a long time to make some music for WW7. I decided only one day before release to scrap what I had been working on and make a whole new song. That new song made it into the final game, but my haste led me to repeat one section for only 3 bars when it should have been 4. Too late to fix it now, I suppose it's one of those fancy time signature things, right?

[13] The MediaPlayer can do this, but it is not actually meant for music in games. It is meant for music player apps. It can't loop segments properly and in some cases has side effects external to the game program.

Chapter 6: Creating content

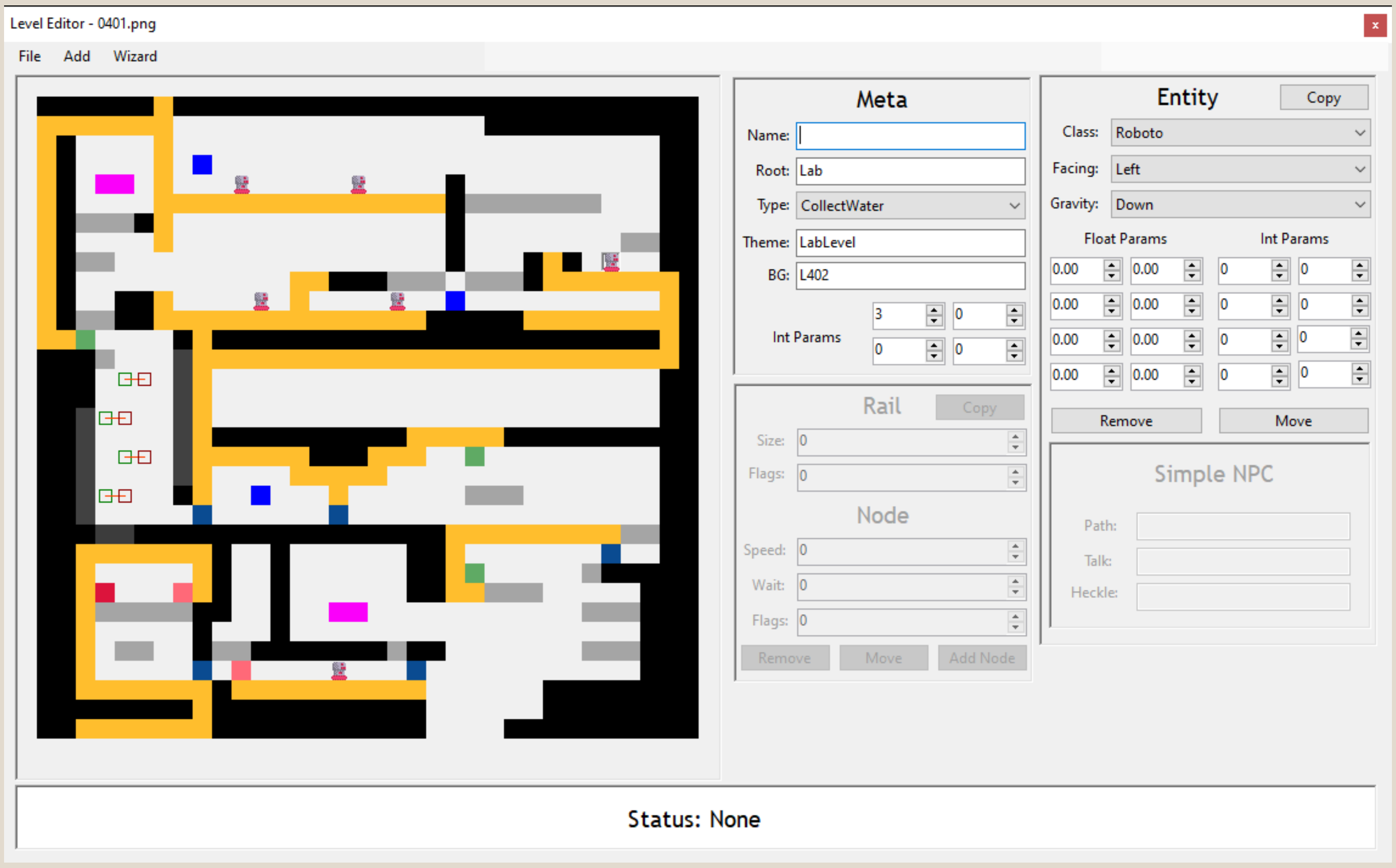
Now that I have established the core concepts of how the game is set up, let's start to look at how content is created for the game. How did I add the many progression for the game? How were new levels defined in data? That sort of thing.

Speaking of levels, let's start there. Each level in the game consists of 3 things, the tilemap(i.e. Level geometry), the entities, and the level properties. The tilemap is defined by a png, which is 36x36 pixels big. Each pixel defines 1 tile, where the color is the tile's type, and the alpha channel encodes a single parameter that can be passed into the tile. This parameter is equal to 255 minus alpha, so that low values can be passed in without the tile becoming nearly invisible. For example, the spikes tiles are color #404040 and the alpha parameter controls the direction they face. The exact mapping can be found in GetTileFromColour(). This meant that I could use paint.NET (an image editor) to edit my levels, which worked out quite well. If you think about it, all the functions like selections, flood fills, brushes, etc are also useful for level editing. You want to move the whole level up one? Ctrl + A -> Up and it's done.



Initially, entities were also spawned in using this system. You can see in the above example, Arnold is spawned by a red pixel. The problem with this is that some entities need more than a single parameter, the alpha channel wasn't enough. I needed to create some auxiliary data that would be paired with the png image file to define the entities. These files are aptly called ".aux" files. These files store all their data in a binary format. It also includes rail data for the moving platforms.

In order to create these binary files, I had to create my own editor. This is the Arid Arnold Level editor, a windows forms application made as a separate project:


Level Editor - 0401.png

File Add Wizard

Meta

Name:
Root: Lab
Type: CollectWater
Theme: LabLevel
BG: L402
Int Params: 3 0 0 0

Rail Copy
Size: 0
Flags: 0

Node
Speed: 0
Wait: 0
Flags: 0
Remove Move Add Node

Entity Copy

Class: Roboto
Facing: Left
Gravity: Down

Float Params		Int Params	
0.00	0.00	0	0
0.00	0.00	0	0
0.00	0.00	0	0
0.00	0.00	0	0

Remove Move

Simple NPC
Path:
Talk:
Heckle:

Status: None

On the left we have our tilemap image. On the right we have all of our properties. The "Meta" section is for properties of the level itself, the folder root, the type of level, the theme of the level, and the background file. We have 4 int parameters that can be passed in. For example, in the "CollectWater" level type, the first parameter is how many bottles we need to collect to complete the level[14].

Then we can place entities on the tile grid using Add->Entity. When you click on an entity it shows you all the parameters you can edit for that entity:

- The type of entity. E.g. Arnold, Robot or
- The initial direction they are facing
- The initial gravity direction.
- 8 float parameters
- 8 int parameters
- The "SimpleNPC" class has special additional properties:
  - The path for their animation data
  - Their dialog options.

The 8 float parameters exist for all types of entities, but most of them do not use all eight. For example, the laser-shooting enemies will read 1 parameter for how often they need to shoot, and another for a timing offset(so they don't all shoot at the same time), the other six parameters are ignored, as are all the int parameters. But for a door, we use the eight int parameters to store the IDs of the levels behind that door.

This isn't the most efficient way to store the entities, but the small space saving from storing only the necessary parameters wasn't worth the hassle of having custom parameters per entity. The only exception

[14] In the final game this was always the total number of bottles in the level, meaning the player always has to collect all the bottles. But originally I had planned it so some levels would have, say, 8 bottles but only require 6 to complete the level. This is why I didn't originally automate it even though I should have by the end.

was for the NPCs. They have three extra string fields that exist only for them. That was because they were added later and I didn't want to break backwards compatibility with the existing aux files.

We also have a panel where we can define rails, which just consist of a connected list of nodes. The nodes have flags, and these flags define if a platform will spawn there or not. This is how we add moving platforms.

In fact, this whole WinForms sub-project is a completely hacky mess. Since I knew it wouldn't have to be that complex I didn't bother with any abstractions or good programming practices. As a result, adding new features just meant packing more and more variables and global state into one big file. Despite its small size, by the time I got around to adding the NPC stuff I was dreading trying to add new features to this mess.

It also sometimes has a weird bug where it "soft crashes". The program seems to be working as normal but pressing "save" doesn't do anything, thus making it completely useless in this state. The only solution is to lose all your unsaved work and restart the program. All in all, not a good program but it got the job done.

Creating levels

So let's walk through the steps for creating new levels. Let's suppose this is the first level in a new world, say world 10. First of all we go to Content/MainCampaign/Levels and copy an existing pair of ".png" and ".aux" files. This makes things easy as they are already set up with the right resolution etc. Each level has its ID as the file name, this would be level 10-1, so the ID is 1001 (first two digits are the world number, and the last two are the level number). The files would be called "1001.png" and "1001.aux". Note that we could pick any number we want, but I just chose this convention to keep things organised.

Next, we need to add a theme to the level. So we go to MainCampaign/Themes and make a new file. A theme is just a remapping of the texture paths so that, for example, the wall texture gets replaced with a special stone texture.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<theme>
    <name>Buk's Cave</name> <!-- Name to be displayed on the UI -->
    <exitColour>18151E</exitColour> <!-- Color of arrows to adjacent rooms -->
    <music>BukCave</music> <!-- Music to be played on this level. -->

    <!-- Texture remappings -->
    <path from="Wall">Cave</path>
    <path from="Platform">CavePlatform</path>
    <path from="CoinFull">GemFull</path>
    <path from="CoinGhost">GemGhost</path>
</theme>
```

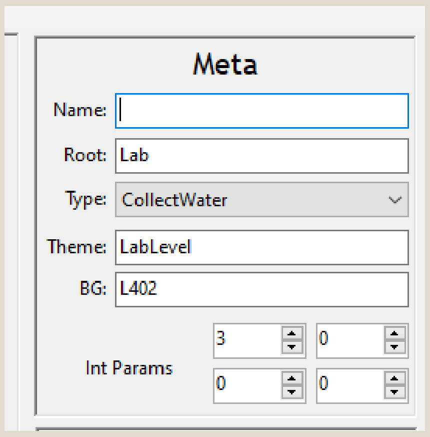
This is an example of how it might look. We remap the wall and platform textures to custom variations, and also remap the coin textures etc. Note that these remapped textures are relative to the "Root" string we input into our ".aux". We then also need to give this level a custom background, go to the Content/BG folder and create a new background. These are just defined by layout files, as discussed above. Now that we have created all the requisite files, we need to point the level data to these files. This is done in the aux editor, in the "Meta" tab.

First we have the name, which is actually never used because the name is also stored in the theme xml file. So we leave that blank.

Next we have the "Root", this is the name of the world within the data. So for example in the theme, when we map "Wall" to "Cave", if the root is "Lab" it will look for the texture "Lab/Cave".

Next we have the theme, give this the same file name as the XML created above.

Then we point it to the BG, remembering this is also relative to the "Root". So here it is set to "L402", meaning it will look for the background "BG/Lab/L402.mlo"

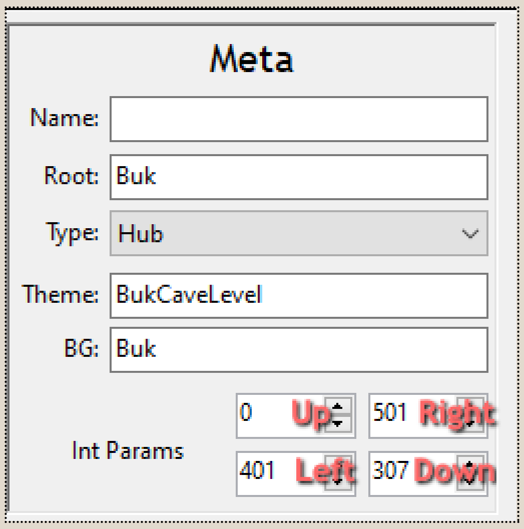

Meta

Name:
Root: Lab
Type: CollectWater
Theme: LabLevel
BG: L402
Int Params: 3 0 0 0

Once we have done that we can place our entities in the level and create the thing. Also, for any further levels with the same theme we can just copy the "png" and "aux" pair without needing to create any more files. If you go into BuildFlags.cs and turn the debug loader on, we can then load directly into this level by changing the debug loader in CampaignManager.cs

```
TimeZoneManager.I.SetCurrentTimeZoneAndAge(0, 0);

//QueueLoadSequence(new HubDirectLoader(901)); //Uncomment this if you want to load a hub level
QueueLoadSequence(new LevelDirectLoader(1001));
```

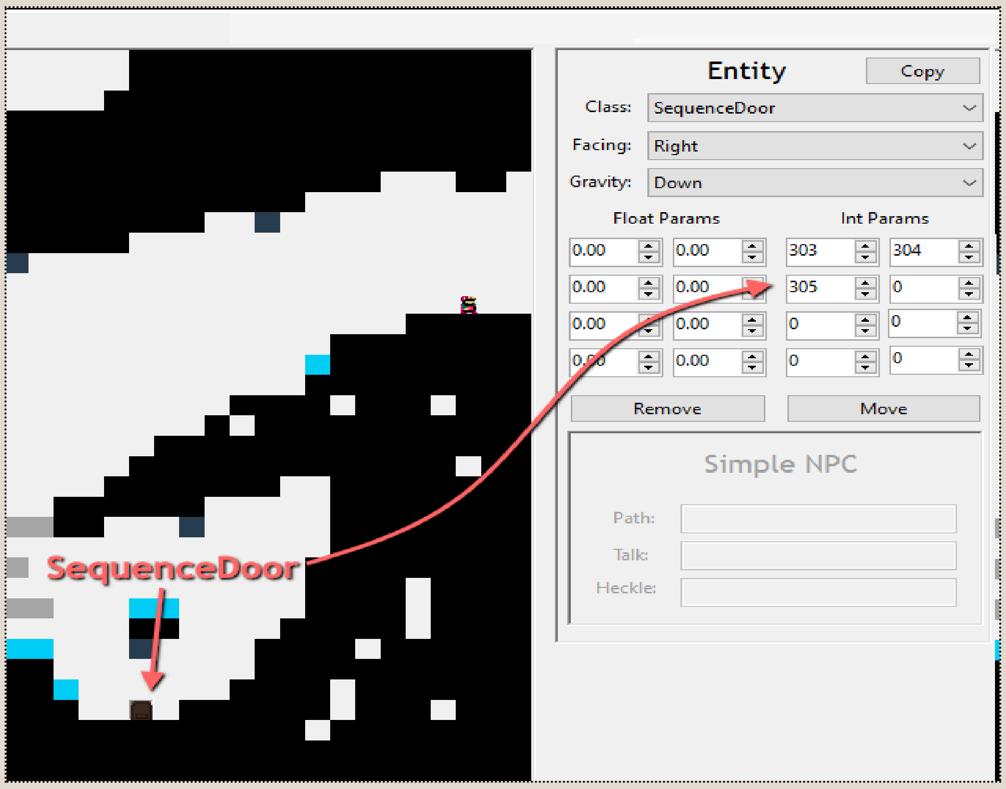
This will let us test the level easily by loading directly into it. Once we are happy with our level, we need to make it accessible from the hub world. Adding a level to the hub world is easy, just do the same process as above but in the "MainCampaign/Hub" folder instead of the "MainCampaign/Levels" folder. Hub levels are attached to one another via the int params in the meta tab.


Meta

Name:
Root: Buk
Type: Hub
Theme: BukCaveLevel
BG: Buk
Int Params: 0 Up, 501 Right, 401 Left, 307 Down

For example, this level shown on the right links to nothing above it, to hub room 501 on the right, 401 on the left, and 307 below it. The type must be set to "Hub" for this to work.

Then within this hub level, we add a "SequenceDoor" entity. These are the doors you enter to access the levels. The 8 int params for these doors correspond to the level IDs that are behind the door.



Entity

Class: SequenceDoor
Facing: Right
Gravity: Down

Float Params: 0.00 0.00 / 0.00 0.00 / 0.00 0.00 / 0.00 0.00
Int Params: 303 304 / 305 0 / 0 0 / 0 0

This sequence door will create a sequence of levels 303 -> 304 -> 305. The player will have to complete them in that order after entering the door.

NPCs

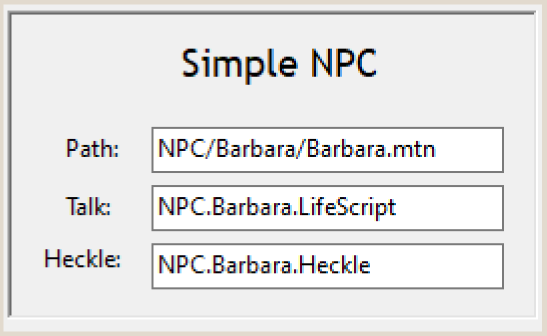
Though all the NPCs in the game have distinct looks and behaviours, they are all the same class: SimpleTalkNPC. The differences in their behaviour is instead defined by the different data they load. This is achieving polymorphism through data rather than with inheritance.

All the NPC data is stored in "Content/NPC", where each folder represents one NPC. Take the "Dok" NPC for example, if we open that folder we will see a few files. The game expects certain files to exist for every NPC, and will look for those within each folder. The first such file is the ".mtn" file, which defines the central properties of the NPC, such as how fast they talk, their text box color and the voice they have when they talk.

```xml
<simpleTalkNPC>
    <name>Dok</name>
    <textStyle>
        <scrollSpeed>0.55</scrollSpeed>
        <framesPerLetter>5</framesPerLetter>
        <fillColor>1E1E28C2</fillColor>
        <borderColor>004705</borderColor>
        <voice>NPC/Voice/AltMale.xml</voice>
    </textStyle>
</simpleTalkNPC>
```

Then the game looks for a file called Idle.mia[15] in the same folder. This encodes an idle animation which plays when the NPC isn't doing anything. Then when the NPC is talking it switches between Default.png for when their mouth is closed, and TalkNormal.png for when their mouth is open. The text scrolls in letter-by-letter, and when the most recent letter is a vowel it will open the NPC's mouth, otherwise it is closed. It also analyses the text for letters of ONLY CAPS, or for exclamation points! This decides if the NPC is angry or not, and if so it will use TalkAngry.png as the "mouth open" texture instead.

When creating a SimpleTalkNPC in the editor, we then specify 3 fields, the "Path" for the look and style of the NPC. Then a "Talk" text handle, which references a localised text handle. There is also a "Heckle" text handle for when the player leaves the NPC before they are done talking.


Simple NPC

Path:	NPC/Barbara/Barbara.mtn
Talk:	NPC.Barbara.LifeScript
Heckle:	NPC.Barbara.Heckle

Text & Strings

When creating a game it is tempting to put your strings as literal values. That is, encoding them directly into the program in quotes. Something like:

```csharp
SimpleTalkNPC myNpc = new SimpleTalkNPC("Hey Arnold, I am speaking!");
```

This is bad because it makes it hard to edit NPC dialog if it is scattered all over the program, and it will also make localisation very messy:

```csharp
SimpleTalkNPC myNpc = new SimpleTalkNPC(IsSpanish() ? "¡Hola Arnold, yo se hablar!" : "Hey Arnold,
I am speaking!");
```

[15] MIA stands for Mono Idle Animation

There is a famous saying: “All problems in computer science can be solved by another level of indirection”. To apply that here, instead of coding the strings directly, we should code a handle that points to a string. This is known as a string handle.

```
SimpleTalkNPC myNpc = new SimpleTalkNPC("Arnold.Greet");
```

Then later I would use the handle to retrieve the string itself when we need to display it:

```
string rawText = LanguageManager.I.GetText(stringID);
```

The reason this is useful is because we can now intercept things in the “GetText” function to load an external file[16]. The handles in “Arid Arnold” are words separated by dots, which represent file paths to text files in the Content/Text folder. For example, “NPC.Barbara.Heckle” points to the file “Content/Text/EN/**NPC/Barbara/Heckle**.txt” if we are playing in English. For other languages, it is the same path except “EN” is swapped out for the language code. Arid Arnold was never localised into any other languages, but if one day I decide to, the process will not require editing any code, instead I can just add an “ES” folder and translate all the “txt” files.

If you look at some of these text files, you might notice some strange symbols. This is because strings need further processing even after they have been localised. The first type of processing is the preprocessor, which can be found in “TextPreprocessor.cs”. A simple example is “InGame.EndingLine3”:

```
Final time: {FinalTime}
```

Here the piece inside the curly braces gets replaced with the current playtime. Note that the files are processed when the string is loaded. This piece of text is loaded at the end of the game and gives the player the playtime of their whole save. We can also substitute other string handles within a preprocessor block. An example of this is that we can include a standard “Press Enter to give water” into all the witch strings:

```
Ahh... so thirsty... Only 4 bottles of water could replenish me.
{Str.NPC.Witch.PromptEnter}
```

This simple system allows us to have programmatic elements to our text. One funny thing I did with this is to give the golden heads random names based off of “Grill Vogel”[17], they would substitute the first letter of each with a random one to create a name like “Brill Yogel” or “Frill Gogel”.

The next level is to format the text with special colors and animation. I can add certain tags that specify color or animations. For some reason I chose Greek characters when I now realise I could have just used English characters as they are already escaped with <> brackets.

```
<Σα[FF0000]050A>HEY! GET BACK HERE!<ψ> I wasn't done talking.
```

In this example, the sigma denotes a shaker animation. A shaker animation can take an optional color, denoted with alpha. Then inside the square brackets is the color, in this case red, and next is the shaker speed and intensity. All the logic for this can be found in ParseControlCharacters(). The Psi(ψ) just resets the character color and gets rid of the shaker effect.

[16] For performance reasons, all the text files are loaded at the program start and kept in a cache. This means no files have to be opened while in-game.
[17] Business hugs with Grill Vogel: https://www.youtube.com/watch?v=80yjtJFf-v0

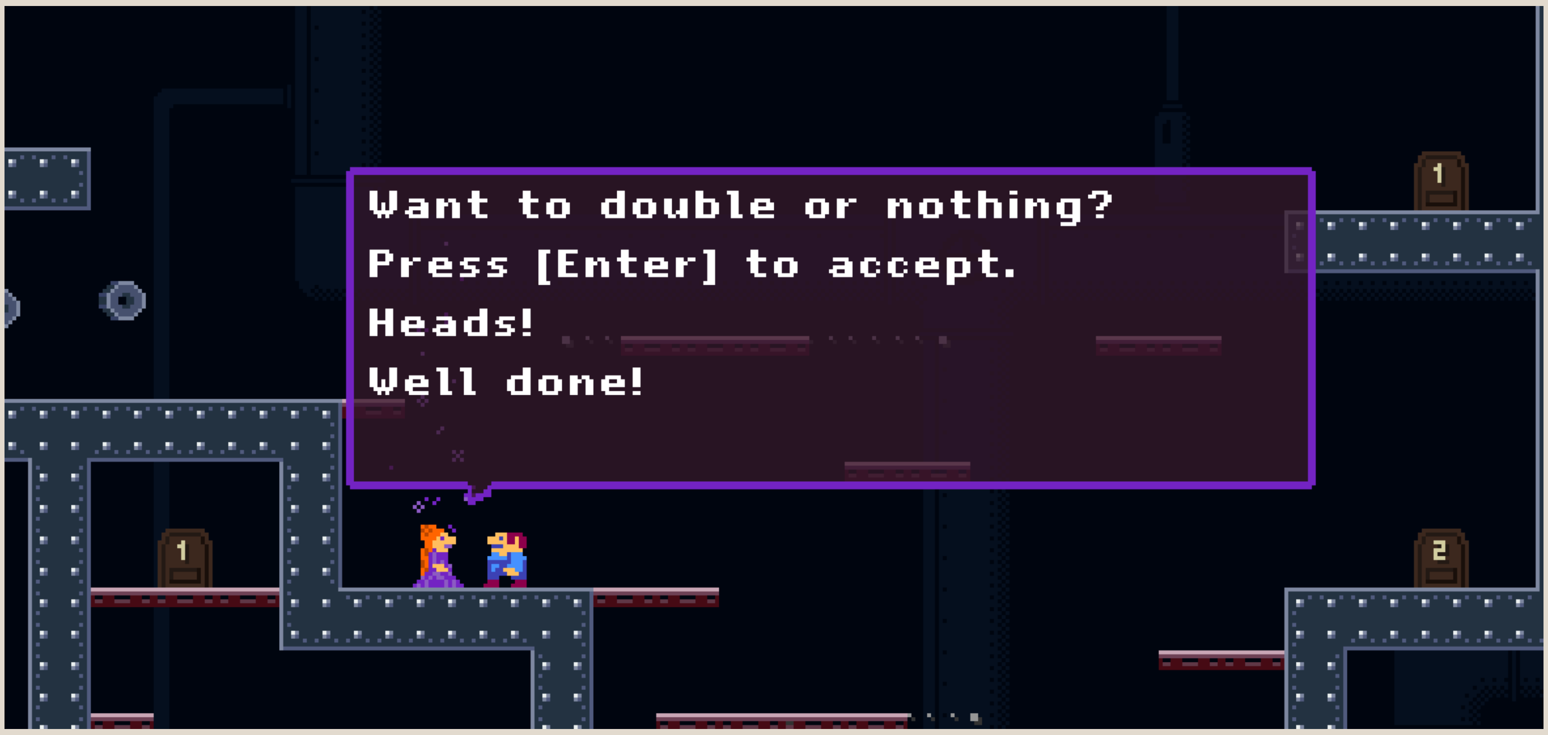
We also have the extra special zeta symbol, which specifies a script. A script declaration might look like this in the text:

```
Want to double or nothing?
{Str.InGame.PromptEnter}
<ζ00>
Well done!
```

So the first two lines will print as normal, and keep in mind the "{Str.InGame.PromptEnter}" will be replaced with the string with handle "InGame.PromptEnter". Then when it reaches the zeta key, it will create a TextScript class instance. There are many scripts, and each one is assigned an ID. In this case we load the script with id 00, which is CoinFlipScript. The IDs are defined by the SCRIPT_TYPES array. This script isn't used in the game, but was made for testing. It will wait for the enter key to be pressed then the script runs a bit of code:

```
protected override void DoOneShot()
{
    MonoRandom rng = RandomManager.I.GetWorld();
    bool heads = rng.PercentChance(50.0f);

    string toAppend = heads ? "Heads!" : "Tails!";
    GetSmartTextBlock().AppendTextAtHead(toAppend);
}
```

The script itself can append text back into the text box. So in game it would look like this:



Want to double or nothing?
Press [Enter] to accept.
Heads!
Well done!

Some scripts can do other things like set flags or give Arnold extra lives. They can also be provided with string arguments. For example:

```
<ζ02[kWaterCollected][0][NPC.Barbara.SteamPlant][NPC.Barbara.SteamPlant]>
```

This calls script 02, which is CheckFlagScript. It is provided with arguments "kWaterCollected", "0", "NPC.Barbara.Steamplant", and "NPC.Barbara.Steamplant".

The system isn't perfect, it was tacked on to a system designed for animating text so it is a bit janky. But it gets the job done. This allowed me to make NPCs that give you key items that are needed to complete the game.

Cutscenes

Cutscenes in the game are encoded as data. There are only two of them, and they can be found in the “Content\Campaigns\MainCampaign\Cinematics” folder. These are very similar to the layout files, except they also encode information in time. At the top of the file we define actors, which are just things we can give a texture and position. Then we just list out instructions for these actors for specific frames. For example:

```xml
<CC_SetActorProps frames="0,7650">
    <actor>BossMan</actor>
    <x>287</x>
    <y>126</y>
    <tex>NPC/BossMan/Default</tex>
</CC_SetActorProps>
```

Here we tell the “BossMan” actor that for frames 0 to 7650, they should be at (287, 126) and display a certain texture. Other commands ask things to move around:

```xml
<CC_CameraLerp frames = "0,300">
    <start>
        <x>-218</x>
        <y>40</y>
        <zoom>4.0</zoom>
    </start>
    <end>
        <x>-218</x>
        <y>-185</y>
        <zoom>4.0</zoom>
    </end>
</CC_CameraLerp>
```

Here we ask the camera to linearly interpolate between two positions. There are also commands to play sounds or display text boxes. By adding several of these commands together we can make the cutscenes.

And yes, I did type these out manually without an editor. It was a lot of trial and error to get the timings right.

Chapter 7: Odds & Ends

There are a few things remaining to cover that don't quite fit into any of the above categories.

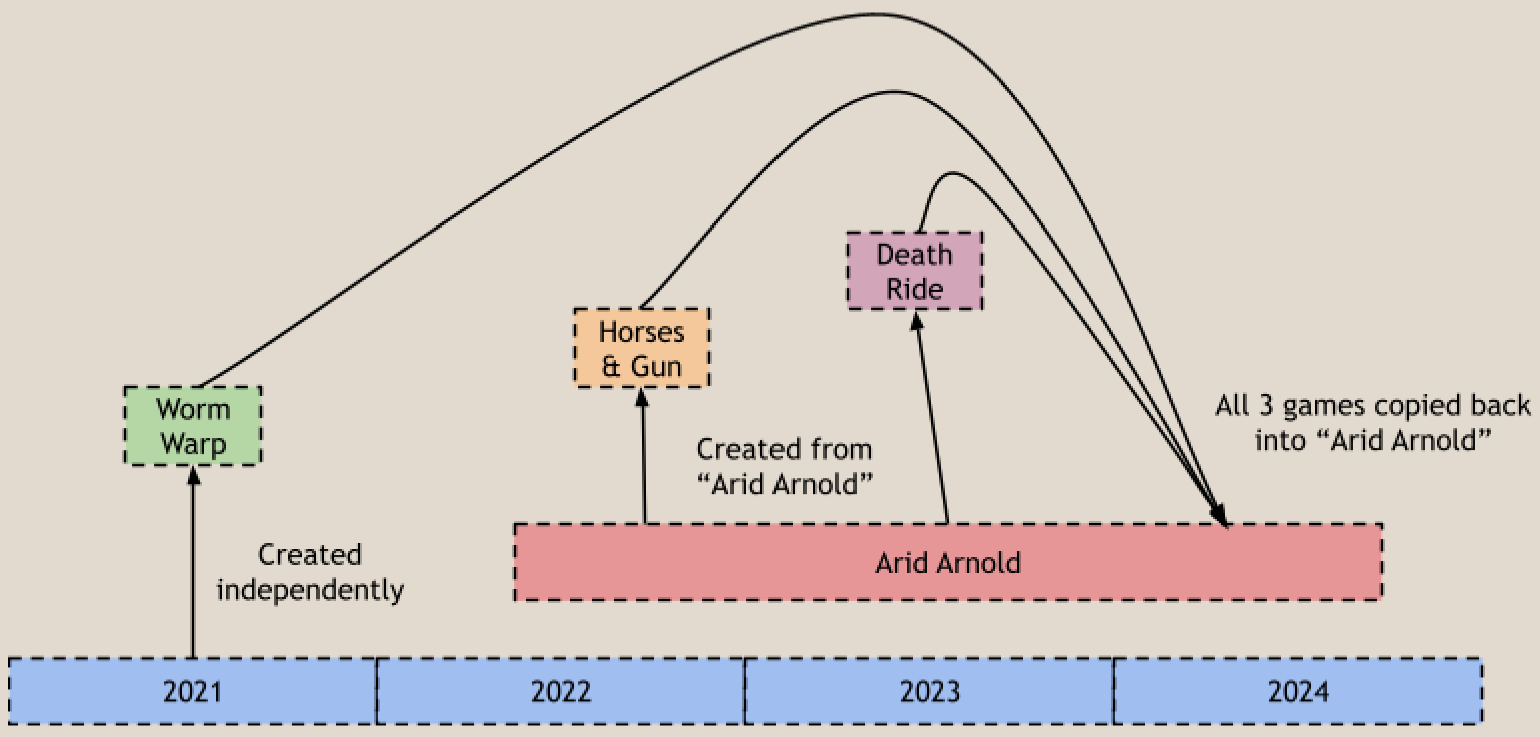
Arcade games

If you have been following along with your own personal copy of the code, you may be wondering why some files are duplicated. For example, why are there three EntityManager.cs files? What is going on inside the Arcade folder?

The story here is a bit less straight-forward than the rest of the code. In "Arid Arnold" there are 3 minigames you can play from these arcade machines: "Worm Warp", "Horses & Gun", and "Death Ride". These games were all independently developed as entries for the GMTK game jam[18]. They were never intended to be part of the "Arid Arnold" codebase. On top of that they were all made in the span of 48 hours, so the code was put together with little regard to so-called "clean code". I would call it something like "decroded code".

The first one of these "Worm Warp", developed in 2021, was my first proper project ever in MonoGame. Although I did have some experience in XNA before that. After that, I made "Horses & Gun" in the summer of 2022, which was only a few months after I had started "Arid Arnold". In order to speed up development, I copied code from the existing version of "Arid Arnold" as the foundation for "Horses & Gun". A year later, in 2023, I did the same thing for "Death Ride", except I was copying newer versions of the files.

Then at some point in 2024, I decided it would be cool to include these games as minigames within "Arid Arnold". Call it scope creep, but this whole process only took me a week to add into the game. I think it was worth it, since it gives the game some meaningful side content where otherwise it would have been completely linear. However, this had the weird effect where I had to copy the code from these arcade games, which were themselves copied from "Arid Arnold", back into the project.



So this is why there are 3 EntityManager classes. The first was made for "Arid Arnold", which then got copied into other projects, which themselves made it back into "Arid Arnold".

[18] Worm Warp: https://augseu.itch.io/worm-warp
Horses & Gun: https://augseu.itch.io/horses-and-gun
Death Ride: https://augseu.itch.io/death-ride-reversal

I had considered cleaning this mess up, but I decided it would be easiest to not mess with these codebases and just treat them as black boxes. Each game was made slightly differently, so I had to standardise the interface for which I could call upon the games, this is done in the ArcadeGame class. I needed to know a few things about the game, to be able to update the game, to know if the game was over, and for the game to be able to send back an output as a frame. Once I could extract that from a game I could then forget about the messy code within. The ArcadeCabinet class then wraps the game and presents the title screen and scoreboard.

For example, look at "DeathRide.cs", this is the bridge between the "Arid Arnold" code and the "Death Ride" code. When we update this class, it calls updates on all the relevant classes within the DeathRide scope. Note because all these classes are duplicates of the "Arid Arnold" classes, it was important to put them in a separate namespace. For example, we wouldn't want to accidentally call the wrong EntityManager while playing "Death Ride", or else Arnold and the bike gangs might be on the screen at the same time.

Then for drawing, it will call on the "Death Ride" screen manager to render out to a RenderTarget2D. This is the output of "Death Ride" stored in a texture. Then the ArcadeCabinet class will composite that with the background and draw it on the "Arid Arnold" level. This is similar to how the RenderTarget2D class is used in the normal mode of operation.

There were a couple of instances where the use of singletons had come back to bite me here. For example, the "Death Ride" game would play music upon startup using its own music manager. So for the sounds and music, I made all the Arcade games use the "Arid Arnold" MusicManager and SFXManager classes instead of their own.

Ghost files

In "Arid Arnold", you can see a replay of your best time as a ghost, similar to how time trials are done in racing games. This is handled in the GhostManager. Every frame, Arnold sends position data to the manager class, where it is added to a big array. The file is then saved using C#'s built-in compression system called the GZipStream.

To play back the files we just read that array from the file when the level starts. We then have a single entity(not controlled by EntityManager) which we force the position and rotation of, and then draw to the screen.

Chapter 8: Final thoughts

“Arid Arnold” was originally created to help me better learn MonoGame. I had wanted to make an RPG game, but decided I should make a simple platformer just to brush up on my skills with MonoGame. Well that simple platformer led me down a two and a half year rabbit hole. But I am glad that I pushed “Arid Arnold” to become the best version of the game I could make it. I am proud that I could finish a game of this scale and not give up halfway through.

On a more technical note, what would I have done differently? Let’s start with the style guide I wrote at the start, annotated with chess move icons:

- 📖 All member variables must be named with an “m” prefix. Known as hungarian notation. E.g. “mMyVariable”
- 📖 All constants must be named in all caps. E.g. PHYSICS_STEPS
- 📖 Files must be logically ordered: constants, members, initialise, update, draw, utils
- ! Use structs instead of tuples
    - Though using tuples is easier, it is much less readable because the members don’t have any name.
- ! All members of a struct must be public
    - Structs are plain-old data. Adding getters and setters is pointless boilerplate.
- 📖 No members of a class can be public
- ! Use inheritance for all “is a” relationships. E.g. Arnold **is a** PlatformingEntity
    - Despite what you head on the internet, inheritance is great and useful.
- ?? Do not use properties, use functions for getter/setters
    - There was no reason to do this, it only added more boilerplate.
- ?? Do not use interfaces, use abstract classes instead
    - Turns out if inheritance is great, so are interfaces! There is a reason they were added to C#.
- ? Do not use lambda functions
    - There were some cases where using lambdas would have been useful, such as for sorting. It is less cumbersome than making a “Sorter” class.
- ?? Do not use optional classes, use null instead. You can use “?” on a struct.
    - I didn’t run into many problems with null, but I can see now why this was a mistake, and why Microsoft has tried to revert it with the nullable-sensitive context[19].
- !! Do not use “var”, manually specify types.
    - Mark my words, type inference is evil. I predict the industry will realise this in about a decade. If I change the return type of a function, I do not want the compiler to automatically fix my code. The idea of some algorithm trying to fix your code without telling you should send shivers down your spine, and that is exactly what type inference is.

Overall I think the code style for “Arid Arnold” is quite good, and it makes the program readable. Even though I forgot how parts of it worked, I could quickly catch up to speed by reading it.

The codebase uses a lot of singletons, and I think this actually was the right decision for my game. It just made everything easier and didn’t really come back to bite me. However, I can see how in a bigger project it could have been a problem, but they are far from the evil the industry purports them to be.

Still though, there are some oddities in how the system is laid out. First of all there are a number of

[19] Nullable reference types: https://learn.microsoft.com/en-us/dotnet/csharp/nullable-references

things in the game that are “things which can be updated and drawn” but are arbitrarily separated into different concepts. For example, what really is the difference between an FX class and an Entity? Both are just things with a position that need updating and drawing every frame. Looking back, I don’t think there was a need for an FXManager and an EntityManager. It’s the same story with a Tile and an Entity, why are the water bottles tiles and not entities? A lot of this could be simplified with the universal concept of a “GameObject”, which is what engines like Unity use.

There’s also 4 different systems designed for positioning objects in a scene. First we have the tile map image which positions tiles and entities. Second, there is the aux file which positions entities and rails. Third is the layout system which positions things from an xml file. Fourth is the cutscene system which positions things in space and time. I’m sure some of this could have been unified. Again, more sane systems like Unity have an editor which is used for all positioning.

While inheritance is great, it may have been a bit overused in this codebase. There are quite a few classes that stretch the concept of “is-a”. For example, LaserBullet inherits from PlatformingEntity. This means that the lasers shooting across the screen have the capability to jump and walk around. The reason I did this is because I wanted them to take the collision logic from the PlatformingEntity class.

I hope this document has been useful to you, and provided an insight into how video games are made. I’m sure you have your own criticisms of the code here, but at the end of the day I was choosing practical solutions over elegant ones. As the codebase grew, and features were added, I had to expand certain systems beyond what they were originally intended to do, which makes for weird code in some places.

If you have questions about “Arid Arnold” or this document, you can contact me via:

Email: IcefishSoftware@gmail.com
Discord: https://discord.gg/qpRbAZkQVh
Bluesky: https://bsky.app/profile/icefish-software.bsky.social

Further Reading

If you liked this document, here are some more things you could read to learn more about MonoGame and game development:

MrGrak, Getting started with MonoGame -
https://rawgit.com/MrGrak/Monogame-Getting-Started/master/index.html

Maddy Thorson, Celeste and Towerfall physics -
https://maddythorson.medium.com/celeste-and-towerfall-physics-d24bd2ae0fc5

Harkonnen, Belts optimisation for Factorio -
https://www.factorio.com/blog/post/fff-176

Frictional Games, In the games of madness -
https://frictionalgames.blogspot.com/

Microsoft, Fundamentals of garbage collection -
https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

Masahiro Sakurai, Creating Games -
https://www.youtube.com/@sora_sakurai_en/videos

Fine Design, BIGNUM BAKEOFF
https://www.youtube.com/watch?v=U1K6TOy6yjU&list=PL-R4p-BRL8NR8THgjx_DW9c92VHTtjZEY

KRAZAM, Microservices -
https://www.youtube.com/watch?v=y8OnoxKotPQ