MONOGAME ROLE-PLAYING GAME DEVELOPMENT

SUCCINCTLY

BY JIM PERRY AND CHARLES HUMPHREY

SUCCINCTLY EBOOK SERIES

Syncfusion®

MonoGame Role-Playing Game Development Succinctly

By
Jim Perry and Charles Humphrey

Foreword by Daniel Jebaraj

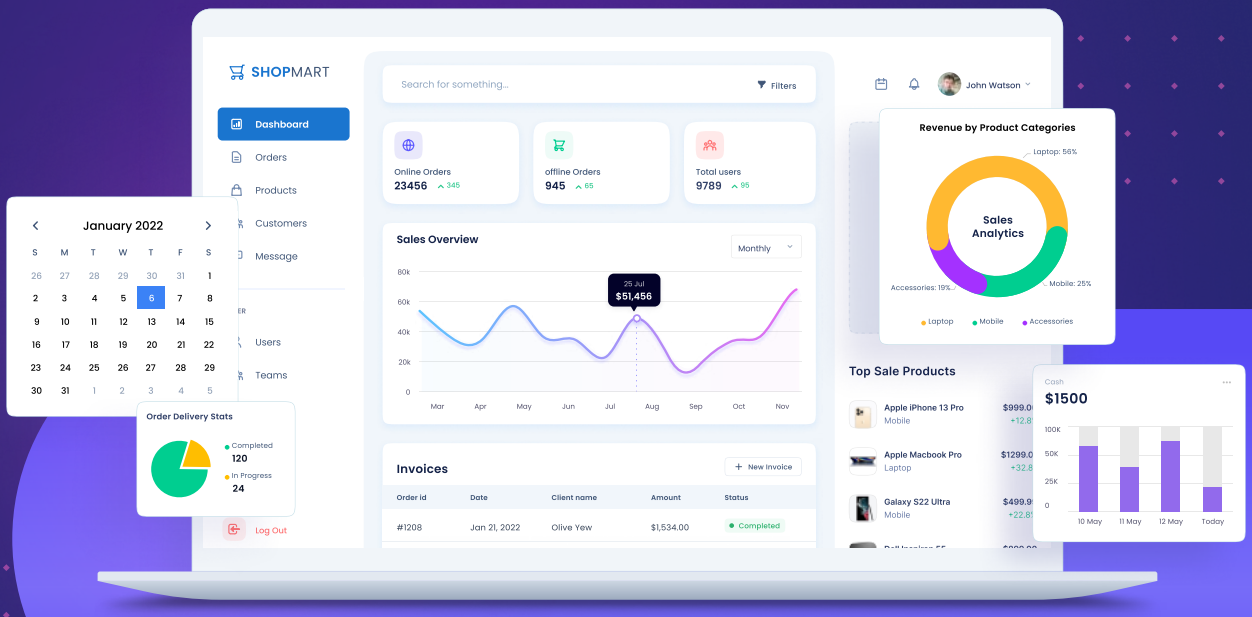
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

ISBN: 978-1-64200-224-9

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, CEO
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Authors

Jim Perry

Jim has been a software engineer for about 20 years, three of which he spent working in the video game industry, contributing to five games during that period. He was a Microsoft MVP for 11 years in both the XNA and Xbox areas. This is his third book on game development.

Charles Humphrey

After working in construction for eight years, Charles took a two-year night school program in C/C++. Charles enjoyed the subject so much, his lecturer offered him a job. That was in 1995, and he hasn't looked back since.

When Microsoft released XNA in 2006, Charles started to learn about programming games and started his Randomchaos XNA blog, hosted on the XNA UK user group website. From 2009–2013, he was recognized as an MVP by Microsoft for his contributions to the community with XNA and DirectX.

Charles tries to remain active on the MonoGame Community boards. He has a public Git repo covering some of his old XNA posts converted to MonoGame, as well as a Patreon page where he's creating a gaming engine along the same lines as Unity from the game up in MonoGame. You can also find him on Twitter.

Who Is This Book For?

This book is intended for people experienced with MonoGame development who are looking to develop a role-playing game in the vein of Diablo. If you don't have experience with MonoGame, there are still a lot of ideas in game design and mechanics that you'll be able to take away, but it's recommended to be comfortable with MonoGame before going any further.

It's assumed that you have MonoGame installed already. If you don't, you can visit the MonoGame site to download it. The current download page has a "Getting Started" section that will help you get started with creating a MonoGame project. Once you're comfortable with everything, you can jump into creating your awesome role-playing game. Head over to the next page, and let's go!

Chapter 1 RPG Basics

What is an RPG?

We'll assume you're reading this because you've heard that role-playing games (RPGs) are an awesome type of video game that a lot of people play, and you want to create one. You may or may not have ever played one, so we'll let Wikipedia help out here with a definition that we can use to start the design process:

> “A role-playing game (abbreviated RPG) is a game in which players assume the roles of characters in a fictional setting. Players take responsibility for acting out these roles within a narrative, either through literal acting, or through a process of structured decision-making regarding character development. Actions taken within many games succeed or fail according to a formal system of rules and guidelines.”

There are some key words/phrases that we'll need to look at here:

- Characters
- Acting out
- Narrative
- Character development
- Formal system of rules

The game usually revolves around a central character, which the player controls. The player makes all the decisions about what the character does, the "acting out" part of the definition. Usually, the character is part of the story (narrative) that is happening and will need to make decisions that may affect the story's end. Along the way, the character will usually become more powerful, gaining or increasing in abilities and physical/mental attributes. These abilities and attributes, along with things like combat, the success or failure of actions the character does, and random events that may occur, all are controlled by the system in the game that is part of the rules that govern the game.

Types of RPGs

Within the RPG genre, there are a number of types of games, each of which adds one or more unique features or types of gameplay that distinguish it from others:

- Action RPG
- Massively multiplayer online RPG (MMORPG)
- Roguelikes
- Tactical RPG
- Sandbox RPG
- First-person, party-based RPG
- Japanese RPG (JRPG)
- Monster tamer

If you have any knowledge of video games, you'll probably have heard of games that fit within these, such as the Fallout series (action and tactical RPG), World of Warcraft (MMORPG), Final Fantasy (JRPG), and Pokémon (monster tamer).

The first step in creating your RPG will be deciding what type of game it will be, as you'll need to design and code the various systems specific to the type and integrate it into the main game.

What makes up an RPG?

While some of the mechanics and systems in RPGs may vary (Skyrim is a very different game than a game in the Fire Emblem series), most share some basics. The player steps into the shoes of one or more characters in a game world, controlling their actions, as the characters make their way through a quest or adventure. Usually things like the character's abilities, skills, and gear will get better along the way until they're powerful enough to defeat the "Big Evil" at the end of the quest.

The difference between RPGs is usually how they enable the character to get there. Some RPGs are melee-based. The Diablo series is a good example. Players usually spend most of the game is a dungeon-like level, fighting off almost endless waves of monsters. Some games have huge open-world environments that the character can wander around in, doing almost anything they want. Skyrim and the Fallout series would be examples of this type of game.

Whatever type of RPG you decide to create, they all have some things in common. The first would be the character(s) the player controls. The character is the player's avatar: the eyes and ears into the game world. As such, the character is usually humanoid, allowing the player to identify with them.

Stats

A character usually has stats that describe the physical attributes of the character. Some typical stats for those familiar with pencil-and-paper RPGs like Dungeons & Dragons (D&D) are:

- **Strength**: This stat measures the raw power a character has, allowing them to do things like lift heavy objects and increase the damage inflicted by melee attacks.
- **Constitution**: This stat measures how hardy the character is, and can influence things like how much damage the character can withstand, or how resistant to poisons they are.
- **Dexterity**: This stat measures things like hand-eye coordination, and possibly how nimble a character is. For more realism and flexibility, you may want to have this stat just measure hand-eye coordination and influence things like accuracy with ranged weapons, or the ability to pick pockets or locks.
- **Agility**: This stat is sometimes used by RPGs to measure overall body nimbleness, influencing things like acrobatics and stealth.
- **Intelligence**: This is the ability to learn something from studying or figuring out things based on your knowledge. It's mainly important to magic-using characters in RPGs.
- **Wisdom**: Depending on your game, this could be intuition or the character's connection to their deity (although I've never really liked this use of the stat). You might not even need something like this for your game, but it's an option that's been a D&D standard for decades.

- **Charisma**: This could be physical beauty or the ability to interact with other entities, or both. This could allow the character to sway non-player characters (NPCs) to do things they normally wouldn't, or to get better bargains from shop owners.

Classes

Most RPGs allow the player to select a class for the character. This is usually the character's profession or what they do for a living. Some typical classes would be fighter, thief, mage, and cleric.

You can have as many or as few classes (or even none) as you need. Many RPG players have a favorite class or style of play, and giving them the ability to play in a way they're used to or enjoy most will make your game more attractive to them.

Race

Most RPGs take place in a fantasy world that has more humanoid types than just regular people. From forest-dwelling elves to monstrous orcs, allowing the player to select a race for their character can completely change the way the player experiences the game, and may influence the way they have to play it.

Your game world may have human towns with people that are fearful of non-humans and therefore bar them from entry or interact with them negatively. Although non-human races may have advantages over humans, this could present problems for the player they wouldn't normally experience if their character was human.

If you have non-human races in your game, think about both the positive and negative aspects the character will have to deal with, as well as how the races compare with each other in terms of stats and abilities. The more options you give the player for customizing their character, the more difficult and time-consuming it will be to balance your game, so be prepared to allow for this in your development process.

We'll examine stats, classes, and races in detail in Chapter 3.

Skills

Skills allow a character to perform some kind of action. This could be attacking a creature, pickpocketing someone in the middle of a crowded city street, sneaking into a house, or climbing the side of a mountain. Anything more difficult than what an unskilled normal being could do would probably require the character to make a skill check. Exactly how this is done is up to you.

Some RPGs have the character "buy" skills with points accumulated by gaining experience in adventuring. Some just allow the character to perform whatever skill they want and have them gradually get better at it.

RPGs that have a buying system could have skills only be usable by certain classes or could make it easier for certain classes to buy skills. Fighters would naturally learn combat-related skills easier and quicker than mages, for example. We'll implement a skill system in Chapter 7.

Magic

The ability for characters to harness magical power is a staple in fantasy RPGs. Even RPGs in a sci-fi universe can have magic; it's just called something different—*psionics*, for example. What's manipulated would be explained in a way that makes sense in that universe, but the result is the same. Characters can do things that appear to be supernatural.

Magic systems can be as basic or complex as you want. You can have multiple types of magic-using classes: a class that manipulates existing matter, a class that can create objects or transform them, or a class that can return dead beings to a semblance of life. However, a magic system is probably the most complex and time-consuming system you'll implement, so plan accordingly.

We'll implement a basic magic system in Chapter 7.

Combat

Eventually, a character is going to come across something or someone they want to fight. Exactly how this plays out can happen in a couple of ways:

- **Real-time**: Time passes in the game exactly as it does in the real world. The player may or may not be able to pause the game to think through the next actions to take.
- **Turn-based**: The player chooses one or more actions to take for a set period of time, usually the equivalent of several seconds. The game may give the player points to spend to take these actions, and actions would cost a set number of points. This is usually restricted to combat where the character can move or attack with a weapon. Each entity involved in the combat waits in a queue for their turn to occur. How the queue is filled is dependent on the rules of the game.
- **Hybrid**: This is a mixture of real-time and turn-based that can vary depending on the game. The game could allow you to switch between both during combat or as a game setting.

We'll implement a combat system in Chapter 9.

World

A character needs to have a place to do all the adventuring. We'll create our game world in Chapter 6.

What’s next

These are just some of the systems that can be in an RPG, and they’re the ones we’ll explore in this book. For now, we’ll take a look at graphics and start to set up a system for having a sprite that represents the character move around on the screen. We’ll also start something to represent the world in which the character will dwell.

Chapter 2 Sprites and Animation

The sprite

A sprite, in the context of this book and computer graphics, is a two-dimensional object that is a part of the current game scene. This could be anything from a floor or scenery tile, to an item in a shop, to our in-game avatar or character.

How will we use them?

As stated previously, a sprite can be a single floor tile, a bush, our in-game avatar, a potion, or an evil orc—pretty much any two-dimensional object representation in our game world. In most of these cases, the sprite is static (unmoving), but in others, this sprite needs to move around the screen or, even while not moving, be animated.

An example of a sprite that may need to move around or across the screen could be an arrow, a magic missile, or our player avatar moving from one place to the next. Again, some of these sprites need animation and others do not. An arrow or spear, for example, will probably have little to no animation, as this object will not have a long time to live on the screen. But our avatar will need to have a number of animations in order to walk, climb, run, and even attack and die. A static sprite could be something like a rock or a campfire, but the latter would likely be animated.

What is sprite animation?

This is where we take a number of textures, or frames, and sequentially draw them one after another in order to give the impression of animation. This is done the same way as old-school cartoons were drawn: one frame or cell at a time.

What is a sprite sheet?

This is a collection of sprite frames all stored on a single texture, or sheet. These frames can be extracted and put into individual keyframes to be used in an animation clip.

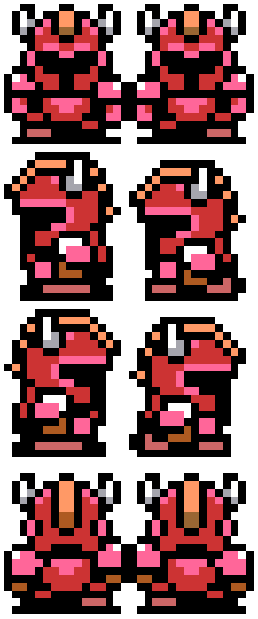


Figure 1: Player sprite sheet

Animation

The code for the animation in this book is from my own MonoGame engine called VoidEngine. I am not sharing all the code from my engine here, but I am sharing snippets of how the animation is handled. You will be able to easily create your own animation classes from these examples, or feel free to use the **`VoidEngineLight`** assembly that comes with the code samples on GitHub in your own projects. If you do, please remember to give us a credit on your project.

What is keyframe animation?

This is how we can specify timings for each frame of our animation frames. This gives us a fair bit of control over the animation. Rather than just move from one frame to the next at a fixed pace, we can use the keyframe to specify how long a single frame is viewed until the animation player moves to the next sprite.

Extracting keyframes from a sprite sheet

There is nothing built into MonoGame to do this—we have had to write our own mechanism. We created a **`SpriteAnimationClipGenerator`** class to do this for us.

To make it work, we need to know the sprite sheet's dimensions and how many frames or slices are in there. We do this by passing them to the constructor of the class.

Code Listing 1: VoidEngineLight – SpriteAnimationClipGenerator

```
public SpriteAnimationClipGenerator(Vector2 spriteSheetDimensions, Vector2
slices)
{
    SpriteSheetDimensions = spriteSheetDimensions;
    Slices = slices;
}
```

We can now use this information to extract animation data, or clips, from the sprite sheet, and we do that in the **`Generate`** method.

Code Listing 2: VoidEngineLight – Generate function

```
public SpriteSheetAnimationClip Generate(string name, Vector2 start,
Vector2 end, TimeSpan duration, bool looped)
```

As you can see from the method signature, we give the clip a name then specify the starting frame location, the last frame's location, the duration of the clip, and whether this frame should be played in a loop.

Now that we have this information, we are going to loop through the sprite sheet and calculate each frame in the animation clip. The first thing we need to do is decide in what direction we need to move in the sprite sheet along the x-axis and y-axis.

Code Listing 3: VoidEngineLight – Generate Function Step 1

```
// Are we going to be moving forward or backwards along the
// X-axis of the sprite sheet to get the animation frames?
if (start.X > end.X)
{
    xIncDec = -1;
    xCnt = (start.X - end.X) + 1;
}
else
    xCnt = (end.X - start.X) + 1;

// Are we going to be moving up or down along the
// Y-axis of the sprite sheet to get the animation frames?
if (start.Y > end.Y)
{
```

```
    yIncDec = -1;
    yCnt = (start.Y - end.Y) + 1;
}
else
    yCnt = (end.Y - start.Y) + 1;
```

We now know how we are going to move through the sprite sheet to calculate the data we need. Before we get going on that, we still need to calculate some more baseline values. We are creating simple animations with this, so we are splitting the time for each frame evenly over each frame in the clip. We calculate the base time for each frame like this:

Code Listing 4: VoidEngineLight – Generate function step 2

```
// This is the base time each frame is made up of.
TimeSpan time = new TimeSpan(duration.Ticks / (long)(xCnt * yCnt));
```

We also want to know the size of each frame on the sheet so we can step to the next frame in the sheet correctly.

Code Listing 5: VoidEngineLight – Generate function step 3

```
// This is the size of a cell on the sprite sheet.
Vector2 cellSize = SpriteSheetDimensions / Slices;
```

Now we can start calculating the values for the frames in this animation clip.

Code Listing 6: VoidEngineLight – Generate function step 4

```
// Is this just one line off the sheet?
// If both start and end Y are the same, then it is a horizontal
// line of keyframes.
if (start.Y == end.Y)
{
    int y = (int)start.Y;
    for (int x = (int)start.X; xCnt > 0; x += xIncDec, xCnt--)
    {
        SpriteSheetKeyFrame frame = new SpriteSheetKeyFrame(new Vector2(x *
cellSize.X, y * cellSize.Y), new TimeSpan(time.Ticks * frameCount++));
        frames.Add(frame);
    }
}
else if (start.X == end.X) // If both start and end X are the same, it's a
vertical slice.
{
    int x = (int)start.X;
    for (int y = (int)start.Y; yCnt > 0; y += yIncDec, yCnt--)
    {
```

```
        SpriteSheetKeyFrame frame = new SpriteSheetKeyFrame(new Vector2(x *
cellSize.X, y * cellSize.Y), new TimeSpan(time.Ticks * frameCount++));
        frames.Add(frame);
    }

}
else // If neither start or end X or Y are the same, then it's a block of
frames.
{
    for (int y = (int)start.Y; yCnt > 0; y += yIncDec, yCnt--)
    {
        float xcnt = xCnt;
        for (int x = (int)start.X; xcnt > 0; x += xIncDec, xcnt--)
        {
            SpriteSheetKeyFrame frame = new SpriteSheetKeyFrame(new
Vector2(x * cellSize.X, y * cellSize.Y), new TimeSpan(time.Ticks *
frameCount++));
            frames.Add(frame);
        }
    }
}
```

First, we check if the start position is on the same vertical as the end position. If it is, then we know we only need to move across the sprite sheet horizontally.

If this is not the case, then we check to see if the start and end positions are the same horizontally, and if they are, we know we only have to move along the sprite sheet vertically.

If neither of these are true, then we are moving diagonally across the sprite sheet.

Regardless of our movement through the sheet, we are calculating each frame's data in the same way with this line:

Code Listing 7: VoidEngineLight – Calculated frame extraction

```
SpriteSheetKeyFrame frame = new SpriteSheetKeyFrame(new Vector2(x *
cellSize.X, y * cellSize.Y), new TimeSpan(time.Ticks * frameCount++));
```

As you can see, we are calculating the frame's position by multiplying the current X and Y positions by the **`cellSize`** **`X`** and **`Y`**. The duration of the frame is calculated using the current frame count and the base time we calculated earlier.

So, that's how we can generate an animation clip.

In our **`GameplayScreen`** **`Activate`** method, we are going to generate our sprite sheet for our simple player avatar.

Code Listing 8: GameplayScreen.cs

```
Texture2D spriteSheet =
_content.Load<Texture2D>("Sprites/Test/TestSheet1");
SpriteAnimationClipGenerator sacg = new SpriteAnimationClipGenerator(new
Vector2(spriteSheet.Width, spriteSheet.Height), new Vector2(2, 4));
```

The first thing we do is load up our sprite sheet from the content pipeline. We can then use the dimensions of this sprite sheet to help set up our **`SpriteAnimationClipGenerator`**. We could put this data into a custom content pipeline and load that up like we do the sprite sheet, but that is a little out of the scope of this book, so we will use our generator.

Once we have an instance of the **`SpriteAnimationClipGenerator`**, we can use it to extract the animations we want from the sheet.

Code Listing 9: Sprite cell definitions

```
Dictionary<string, SpriteSheetAnimationClip> spriteAnimationClips = new
Dictionary<string, SpriteSheetAnimationClip>()
{
    { "Idle", sacg.Generate("Idle", new Vector2(1, 0), new Vector2(1, 0),
new TimeSpan(0, 0, 0, 0, 500), true) },
    { "WalkDown", sacg.Generate("WalkDown", new Vector2(0, 0), new
Vector2(1, 0), new TimeSpan(0, 0, 0, 0, 500), true) },
    { "WalkLeft", sacg.Generate("WalkLeft", new Vector2(0, 1), new
Vector2(1, 1), new TimeSpan(0, 0, 0, 0, 500), true) },
    { "WalkRight", sacg.Generate("WalkRight", new Vector2(0, 2), new
Vector2(1, 2), new TimeSpan(0, 0, 0, 0, 500), true) },
    { "WalkUp", sacg.Generate("WalkUp", new Vector2(0, 3), new Vector2(1,
3), new TimeSpan(0, 0, 0, 0, 500), true) },
};
```

We now have five animation clips we can use to animate our player avatar, but how do we do that? All we have at the moment are clips—how do we play them?

Animation player

We can now generate animation clips, but for any given screen entity, we may have a number of animations that we want to play. Our player avatar, for example, will want to be able to walk left, right, up, and down, and take other actions. So, we need to store all these possible animations in one place and be able to play them as we need them. That's where our **`SpriteSheetAnimationPlayer`** comes in.

We have a few properties in this class.

Code Listing 10: VoidEngineLight – Animation player

```
public TimeSpan AnimationOffSet { get; set; }
```

```
protected bool _IsPlaying = false;
public bool IsPlaying { get { return _IsPlaying; } }

public Vector2 CurrentCell { get; set; }

public int CurrentKeyframe { get; set; }

public event AnimationStopped OnAnimationStopped;



protected SpriteSheetAnimationClip currentClip;
public SpriteSheetAnimationClip CurrentClip
{
    get { return currentClip; }
}

TimeSpan currentTime;
public TimeSpan CurrentTime
{
    get { return currentTime; }
}

public Dictionary<string, SpriteSheetAnimationClip> Clips { get; set; }
```

TimeSpan AnimationOffSet

To offset the timing of the animation, we can use this when we have several animated sprites on the screen at the same time. Rather than have them play the same frames at exactly the same time, we can offset each of them so they look a little more natural. A hallway with flaming torches is a good example: we would not want each flame playing the same animation at the same time.

bool IsPlaying

This indicates if the animation player is currently playing a clip. It's handy if you want to trigger some environmental event when the animation is playing.

Vector2 CurrentCell

This is the current cell in the current clip. This is the bit we really need in order to extract the right frame from the sprite sheet, and it includes the coordinates for the frame we need to draw.

int CurrentKeyframe

This may be useful if we want to play a sound when a specific frame is reached.

event AnimationStopped OnAnimationStopped

This event can be subscribed to and is triggered when an animation stops. This is great for looping clips, and you can use this to chain one animation clip after another.

SpriteSheetAnimationClip CurrentClip

This is the clip that is currently in use or is ready to be used.

TimeSpan CurrentTime

This is the current time in the clip being played, and it will range from zero to the duration of the clip.

Dictionary<string, SpriteSheetAnimationClip> clips

This is the container for all the clips we will want to play for a given sprite sheet. We can use this to move from one clip, let's say from **`Idle`** to **`WalkLeft`**.

The constructor for the **`SpriteSheetAnimationPlayer`** is pretty simple: we just pass it the clips we have just generated and the time offset we want to use.

Code Listing 11: VoidEngineLight – Animation player constructor

```
public SpriteSheetAnimationPlayer(Dictionary<string,
SpriteSheetAnimationClip> clips = null, TimeSpan animationOffSet = new
TimeSpan())
{
    AnimationOffSet = animationOffSet;
    Clips = clips;
}
```

We can now use the **`StartClip`** and **`StopClip`** to—you guessed it—start and stop the animation clips.

StartClip

Code Listing 12: Animation Player StartClip

```
public void StartClip(string name, int frame = 0)
{
    StartClip(Clips[name]);
}

public void StartClip(SpriteSheetAnimationClip clip, int frame = 0)
{
    if (clip != null && clip != currentClip)
    {
        currentTime = TimeSpan.Zero + AnimationOffSet;
        CurrentKeyframe = frame;
```

```
        currentClip = clip;

        _IsPlaying = true;
    }
}
```

As you can see, we have an overloaded function for **`StartClip`** since we may wish to start a clip by name, or if we already have the clip, just pass and use that.

The first thing we do is ensure the clip we are working with is valid—that is to say it's not null and it's not the clip we are currently working with. We then set the current time to zero, plus any time offset we want to use. We set the frame to the frame we want to use, the current clip to the one passed in, and finally, set **`IsPlaying`** to **`true`**. This sets up our clip to be played, and will kick off the logic in our **`Update`** method (we will come back to that in a little while).

StopClip

Code Listing 13: Animation player StopClip

```
public void StopClip()
{
    if (currentClip != null && IsPlaying)
    {
        _IsPlaying = false;

        if (OnAnimationStopped != null)
            OnAnimationStopped(currentClip);
    }
}
```

As with **`StartClip`**, the first thing we do is check the clip we are working with and check that we are currently playing. If we have a valid clip and we are indeed playing, we simply set **`IsPlaying`** to **`false`**, and if anything is subscribed to our **`OnAnimationStopped`** event, we let it know it's stopped.

Update

Code Listing 14: Animation player Update

```
public void Update(TimeSpan time)
{
    if (currentClip != null)
        GetCurrentCell(time);
}
```

This is almost where the magic happens. Again, we check whether the current clip is valid. If it is, then we call **`GetCurrentCell`**, passing the elapsed game time, and this is where all the work is done.

GetCurrentCell

Code Listing 15: Animation player GetCurrentCell

```
protected void GetCurrentCell(TimeSpan time)
{
    time += currentTime;

    // If we reached the end, loop back to the start.
    while (time >= currentClip.Duration)
        time -= currentClip.Duration;

    if ((time < TimeSpan.Zero) || (time >= currentClip.Duration))
        throw new ArgumentOutOfRangeException("time");

    if (time < currentTime)
    {
        if (currentClip.Looped)
            CurrentKeyframe = 0;
        else
        {
            CurrentKeyframe = currentClip.Keyframes.Count - 1;
            StopClip();
        }
    }

    currentTime = time;

    // Read keyframe matrices.
    IList<SpriteSheetKeyFrame> keyframes = currentClip.Keyframes;

    while (CurrentKeyframe < keyframes.Count)
    {
        SpriteSheetKeyFrame keyframe = keyframes[CurrentKeyframe];

        // Stop when we've read up to the current time position.
        if (keyframe.Time > currentTime)
            break;

        // Use this keyframe.
        CurrentCell = keyframe.Cell;

        CurrentKeyframe++;
    }
}
```

As you can see, this is the workhorse of the **SpriteSheetAnimationPlayer** class. The first thing we do is add the clip's current time to the elapsed time passed in. If that time value is longer than or equal to the clip duration, then we need to set the time back to the start of the clip.

If our time ends up being less than zero or greater than the clip duration, we need to throw a controlled exception to indicate there is an issue with the time. If we left it to run, we would get an **`ArgumentOutOfRangeException`** thrown when we try to get the current cell later, so we might as well know about it sooner rather than later.

If time is now less than the current time, then we must have reached the end and looped back to the start. So, we check if this clip is looped, and then we just set the current frame to 0 so we can start the animation again. If it's not, then we set the current frame to the last frame in the clip and call the **`StopClip`** function. As we saw in the previous code listing, this will stop the clip and inform any subscribers to the event that it has stopped.

If we've come this far, we are still playing the animation clip. We get a list of the keyframes from the current clip and loop through them to find the current cell coordinates. We do this by checking if the keyframe's time is greater than the current time. If it is, then the last frame is the frame we are currently on, so we break out of the loop here. If it isn't, then this frame could be the frame we are on, so we store it and move on the current keyframe.

Animation clips

In both the **`SpriteAnimationClipGenerator`** and the **`SpriteSheetAnimationPlayer`** we have spoken about animation clips, but what do they look like in code?

Code Listing 16: VoidEngineLight – Animation clip

```
public class SpriteSheetAnimationClip
{
    public string Name { get; set; }
    public bool Looped { get; set; }
    public TimeSpan Duration { get; set; }

    public List<SpriteSheetKeyFrame> Keyframes { get; set; }

    public SpriteSheetAnimationClip() { }

    public SpriteSheetAnimationClip(string name, TimeSpan duration,
List<SpriteSheetKeyFrame> keyframes, bool looped = true)
    {
        Name = name;
        Duration = duration;
        Keyframes = keyframes;
        Looped = looped;
    }

    public SpriteSheetAnimationClip(SpriteSheetAnimationClip clip)
    {
        Name = clip.Name;
        Duration = clip.Duration;

        SpriteSheetKeyFrame[] frames = new
SpriteSheetKeyFrame[clip.Keyframes.Count];
        clip.Keyframes.CopyTo(frames, 0);
```

```
        Keyframes = new List<SpriteSheetKeyFrame>();
        Keyframes.AddRange(frames);

        Looped = clip.Looped;
    }
}
```

They are pretty much just a storage class for the data required for a given clip, including a list of all its keyframe data.

Keyframes

Code Listing 17: VoidEngineLight – Keyframe

```
public class SpriteSheetKeyFrame
{
    public Vector2 Cell { get; set; }
    public TimeSpan Time { get; set; }

    public SpriteSheetKeyFrame() { }
    public SpriteSheetKeyFrame(Vector2 cell, TimeSpan time)
    {
        Cell = cell;
        Time = time;
    }
}
```

Again, this is a storage class for the keyframe data. The **`Cell`** gives the starting X and Y position in the sheet and the **`Time`** gives the duration the frame is displayed.

Playing animation clips

We now have a set of animation clips and an animation player that can give us the data at runtime in order to draw the cell we want as an animation plays, but we have no way of bringing these together.

If we return to our **`GameplayScreen`** class and the **`Activate`** method, we can see we have a **`Sprite`** class.

Code Listing 18: GameplayScreen.cs

```
playerAvatar = new Sprite(spriteSheet, new Point(32, 40), new Point(16,
20));
playerAvatar.animationPlayer = new
SpriteSheetAnimationPlayer(spriteAnimationClips);
playerAvatar.StartAnimation("Idle");
```

This code initializes our player avatar, passing in the sprite sheet we used to generate the animation clips, the size we want to render our player at, and the physical cell size in pixels. We then give it an instance of the **`SpriteSheetAnimationPlayer`** populated with our extracted animation clips, and finally, tell it so start the **`Idle`** animation clip.

Animation in action

Sprite class

Code Listing 19: Sprite.cs

```
public class Sprite
{
    public Vector2 Position { get; set; }
    public Point CellSize { get; set; }
    public Point Size { get; set; }
    public Texture2D spriteTexture { get; set; }
    protected SpriteSheetAnimationPlayer _animationPlayer;
    public SpriteSheetAnimationPlayer animationPlayer
    {
        get { return _animationPlayer; }
        set
        {
            if (_animationPlayer != value && _animationPlayer != null)
                _animationPlayer.OnAnimationStopped -= OnAnimationStopped;

            _animationPlayer = value;
            _animationPlayer.OnAnimationStopped += OnAnimationStopped;
        }
    }

    public Color Tint { get; set; }

    protected Rectangle sourceRect
    {
        get
        {
            if (animationPlayer != null)
                return new Rectangle((int)animationPlayer.CurrentCell.X,
(int)animationPlayer.CurrentCell.Y, CellSize.X, CellSize.Y);
            else
            {
                if (CellSize == Point.Zero)
                    CellSize = new Point(spriteTexture.Width,
spriteTexture.Height);
```

```
                return new Rectangle(0,0, CellSize.X, CellSize.Y);
            }
        }
    }

    public Sprite(Texture2D spriteSheetAsset, Point size, Point cellSize)
    {
        spriteTexture = spriteSheetAsset;
        Tint = Color.White;
        Size = size;
        CellSize = cellSize;
    }

    protected virtual void OnAnimationStopped(SpriteSheetAnimationClip
clip)
    {
        return;
    }

    public virtual void StartAnimation(string animation)
    {
        if (animationPlayer != null)
            animationPlayer.StartClip(animation);
    }

    public virtual void StopAnimation()
    {
        if (animationPlayer != null)
            animationPlayer.StopClip();
    }

    public virtual void Update(GameTime gameTime)
    {
        if (animationPlayer != null)
            animationPlayer.Update(gameTime.ElapsedGameTime);
    }

    public virtual void Draw(GameTime gameTime, SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(spriteTexture, new Rectangle((int)Position.X,
(int)Position.Y, (int)Size.X, (int)Size.Y), sourceRect, Tint);
    }
}
```

The **`Sprite`** class is used as the base class for all our in-game sprites. It has a number of properties:

Vector2 Position

This is the position of the sprite on the screen.

Point CellSize

This is the physical size of a given cell in our sprite sheet in pixels.

Point Size

This is the size in pixels we want to draw our sprite.

Texture2D spriteTexture

This is our sprite sheet texture used to render our sprite.

SpriteSheetAnimationPlayer animationPlayer

This is a populated instance of the **`SpriteSheetAnimationPlayer`** we discussed previously. When we add an animation player, we wire up to its **`OnAnimationStopped`** event. If we already have a player, we unwire ourselves from its **`OnAnimationStopped`** event.

Color Tint

We can use this to “tint” the color of our sprites if we want.

Rectangle sourceRect

This is what we use to know what part of the sprite sheet we want to render. If there is no animation player, then we assume that it must be the whole texture we want to render. If there is an animation player, then we use the cell size that we have been given to set up its height and width.

The constructor for the class is simple enough. We pass in the sprite sheet, the size we want to render, and the cell size in the sprite sheet and store them for later use.

OnAnimationStopped

While not currently implemented here, it may be by derived classes.

StartAnimation

This method will start a given animation clip if we have an animation player.

StopAnimation

This function will stop an animation clip if we have an animation player.

Update

This method, if we have an animation player, calls its **`Update`** method.

Draw

Finally, we get to draw our sprite using the texture provided, set the destination rectangle on the screen, use our calculated source rectangle, and tint the sprite based on the tint color we have.

Moving our character

We now have all the elements in place for us to be able to animate and move our player avatar. Return to the **`GameplayScreen`** class; this time we will go into its **`HandleInput`** method.

Code Listing 20: GameplayScreen.cs HandleInput

```
if (input.IsKeyPressed(Keys.Down, ControllingPlayer, out player))
    playerAvatar.animationPlayer.StartClip("WalkDown");
else if (input.IsKeyPressed(Keys.Up, ControllingPlayer, out player))
    playerAvatar.animationPlayer.StartClip("WalkUp");
else if (input.IsKeyPressed(Keys.Left, ControllingPlayer, out player))
    playerAvatar.animationPlayer.StartClip("WalkLeft");
else if (input.IsKeyPressed(Keys.Right, ControllingPlayer, out player))
    playerAvatar.animationPlayer.StartClip("WalkRight");
else
    playerAvatar.animationPlayer.StartClip("Idle");
```

Here, if the screen is not paused, we can specify what animation we want to have played by our player avatar. We can press the Down arrow key to play the **`WalkDown`** animation, the Up arrow key to play the **`WalkUp`** animation, the Left arrow key to play the **`WalkLeft`** animation clip, and the Right arrow key to play the **`WalkRight`** animation clip. If there is no input, we play the **`Idle`** animation.

Now, in the **`Update`** method in the **GameplayScreen** class we can move our animated player avatar based on the animation being played.

Code Listing 21: GameplayScreen.cs update

```
float translateSpeed = 0.5f;

switch (playerAvatar.animationPlayer.CurrentClip.Name)
{
    case "WalkDown":
        if (!currentLevel.IsSolid(playerAvatar.Position + new Vector2(0,
translateSpeed)))
            playerAvatar.Position += new Vector2(0, translateSpeed);
        break;
    case "WalkLeft":
```

```
        if (!currentLevel.IsSolid(playerAvatar.Position + new Vector2(-
translateSpeed, 0)))
            playerAvatar.Position += new Vector2(-translateSpeed, 0);
        break;
    case "WalkRight":
        if (!currentLevel.IsSolid(playerAvatar.Position + new
Vector2(translateSpeed, 0)))
            playerAvatar.Position += new Vector2(translateSpeed, 0);
        break;
    case "WalkUp":
        if (!currentLevel.IsSolid(playerAvatar.Position + new Vector2(0, -
translateSpeed)))
            playerAvatar.Position += new Vector2(0, -translateSpeed);
        break;
    case "Idle":
        break;
}
```

If the screen is active, we call our player avatar's **`Update`** method, set a translation speed, and then look to see what animation clip is being played. If the **`WalkDown`** animation is playing, then the player avatar moves down the screen. **`WalkLeft`** will move our player avatar to the left, and **`WalkRight`** moves the player avatar to the right. When the **`WalkUp`** animation is playing—you guessed it—the avatar is moving up the screen.

Finally, we do a quick check to make sure that our player avatar can't escape the screen.

What's next

We now have a framework for creating and animating sprites within our game, whether that sprite is our player, an NPC, or even the environment and equipment. Now we need a framework for creating our player characters and giving them characteristics, skills, and classes.

Chapter 3 Character Creation

Stats

The first thing we'll look at as part of the character creation process is an entity's stats. We use the term *entity* since not only does the player's character have stats, but most beings (human and otherwise) will have them too. The stats we'll use for our character will be pretty standard. If you've played Dungeons & Dragons, you'll recognize most of them immediately:

- Strength
- Dexterity
- Agility
- Constitution
- Intelligence

The only non-D&D stat we have is agility. In D&D, dexterity is used for both hand-eye coordination and full-body skills. This isn't the most flexible or realistic way to handle this. By dividing this up into two stats, we give ourselves more flexibility.

Dexterity will be used for hand-eye related skills, and agility will be used for things like acrobatics and other full-body skills.

For those not experienced with RPGs, here's a brief description of the other stats:

- **Strength** is used for helping to calculate the damage a character can inflict in hand-to-hand combat and how much a character can carry.
- **Constitution** determines how much damage a character can take before dying, and how quickly a character recovers from injury and other ailments (poison, for example).
- **Intelligence** is used for learning skills; in this case, it's mainly for magic-using characters.

For the veteran RPGers, you may be wondering where charisma is. For our small demo, there's really no need for it. It's useful for things like bargaining with shop owners to get a better price and is an important stat for professions like bards. We won't be dealing with anything that would require it. If you add features like this to your game, feel free to add it and work it in. It won't be much different from implementing the other stats.

We'll use two classes to handle our stats: a base class for general properties of a stat, and one specific to entities.

Code Listing 22: Stat classes

```
public enum StatType
{
    Regular,
    Calculated
}
```

```
public class Stat
{
    public StatType Type { get; set; }
    public string Name { get; set; }
    public string Abbreviation { get; set; }
    public string Description { get; set; }

    public string StatCalculation { get; set; }

    private static readonly char[] operators = { '+', '-', '*', '/' };

    public const short MaxValue = (short)DieType.d100;
    public const int PoundsPerStatPoint = 3;
}

[Serializable]
public class EntityStat
{
    private short Value;
    public string StatName { get; set; }
    public short BaseValue { get; set; }
    public List<Bonus> Bonuses { get; set; }

    public EntityStat()
    {
    }

    public EntityStat(string stat, short value)
    {
        StatName = stat;
        Value = value;
    }

    public short CurrentValue
    {
        get
        {
            short val = Value;

            if (Bonuses != null)
            {
                foreach (Bonus bonus in Bonuses)
                    val += bonus.Amount;
            }

            return val;
        }

        set { }
```

```
    }

    public void Update(float time)
    {
      if (Bonuses == null)
            return;

        Bonus bonus;

        for (int i = Bonuses.Count - 1; i >= 0; i--)
        {
            bonus = Bonuses[i];

            bonus.ElapsedTime += time;

            if ((int)bonus.ElapsedTime >= bonus.Duration)
                Bonuses.Remove(bonus);
            else
                Bonuses[i] = bonus;
        }
    }
}
```

Note: The `DieType` values represent the physical dice used in pencil-and-paper RPGs. Such dice have different numbers of sides, with 4, 6, 8, 10, 12, 20, and 100 being the usual types.

While we won't be doing this for our sample game, there could be situations where you want to have a stat that's calculated based on other stats or some external information. We've included a way for the stats to be differentiated by using an enum type.

The members of the **`Stat`** class are straightforward. There will probably be times (usually in the UI somewhere) where you don't want to (or can't) display the entire stat name, so you'll need an abbreviation. We'll use three letters for each: **`STR`**, **`DEX`**, **`AGI`**, **`CON`**, and **`INT`**.

For calculated stats, the pieces that make up the calculation are stored in a string that will have to be parsed using the **`operators`** array. You would split the string based on the elements in the array, get the values of the elements, and calculate the total.

Stats and other numeric information that make up an entity usually have a minimum and maximum value. We're assuming **`1`** for the minimum, and we'll use a constant for the maximum, which we cast from one of our **`DieType`** (as in dice, not death) enum values. We'll have a lot of numeric data of various possible maximums, some randomly generated. Having an enum with the possible maximums allows us to easily tweak this data to balance the game. If something has too high a maximum, we just replace it with the next-lowest enum value. This should be sufficient for our needs.

The **EntityStat** class is a little more complex since nearly everything about an entity can be modified during the course of the game if you want it to be. In a fantasy RPG, magic can do just about anything. In a sci-fi RPG, advanced technology is the substitute for magic.

Value is the current value of the stat without taking bonuses into account. There are functions for increasing or decreasing this value, such as when the character gains a level and their stats increase. There are times when a stat could permanently decrease as well.

Bonuses that affect a stat are added using the **AddBonus** function, and the **Update** function is called every frame—although this could be changed to every second or whatever works for your game.

The **CurrentValue** function takes all the bonuses into account, both positive and negative, as it returns the bonus-adjusted value.

There are some additional functions for modifying the **Value** member and adding bonuses in the sample project that aren't listed here.

The **Bonus** class that's used here and in other places is very simple:

Code Listing 23: Bonus class

```
public enum BonusType
{
    Disease,
    Magic,
    Poison,
    Potion,
    Other
}

public class Bonus
{
    public BonusType Type;
    public short Amount;
    public int TimeStarted;
    public int Duration;
    public float ElapsedTime;
}
```

Since there might be counters to some types of negative bonuses, we need to be able to tell what the different types of bonuses are; thus, we have the **Type** member.

The **Duration** member is the number of seconds that the bonus lasts. You could have a permanent bonus. Setting this member to **0** or **-1** could allow you to tell that it's a permanent bonus. You would need to have additional code in the check for this before checking the value against the **ElapsedTime** member.

The **`ElapsedTime`** is a simple **`float`** to give some flexibility in values that an **`int`** wouldn't. Something like the **`TimeSpan`** structure, which is often used for tracking times, is more overhead than is needed for our simple game.

Races

Races in RPGs usually give the player the ability to tailor their character a bit to fit the style of play they like. Most of the time, members of a race will have a specific skill or ability they're better at than other races. We'll go over a couple here, and you can add as many to your game as you need:

- **Human**: Humans are usually okay at everything, exceptional at nothing. This allows the player to do just about anything in the game and have a decent chance of success. They'll rarely be a master of anything until close to the end of the game, and then only if they dedicate themselves to that area.
- **Elf**: There are several kinds of elves in most fantasy lore, but they're all usually good at a couple of things. Ranged combat with a bow and spellcraft are the usual areas of expertise. Elves are usually as excellent at sneaking, tracking, and moving through wooded areas.
- **Dwarf**: Dwarves are renowned for their constitution and hardiness, as well as their prowess in hand-to-hand combat. This makes them excellent fighters and often resistant to poisons and other things that would usually kill non-dwarves.
- **Halfling**: Halflings are usually thieves in the fantasy realm. Their smaller size usually makes them well qualified to sneaking around and getting into places other races would find difficult.

These four races will give our small game a good range of choices for a player.

We'll represent the differences in races by adjusting the character's stats a bit as shown in the following table:

Table 1: Stat modifications by race

	Strength	Dexterity	Agility	Constitution	Intelligence
Human	0	0	0	0	0
Elf	-2	+1	+1	-1	+1
Dwarf	+2	-1	-1	+1	-1
Halfling	-1	+1	+2	-1	-1

You'll probably want to have a race that has no modifications—partly to balance the other races against, and partly to have a baseline for the game itself. If you can play through the game as a human appropriately for each of your difficulties, you can then compare that playthrough to the other races.

Notice that the positives and negatives balance out so that no race has an obvious advantage over another. If you don't do this, players will almost inevitably either exploit the imbalance or get angry that the imbalance exists (or both).

The class we'll use to hold our race data is fairly simple and can be expanded as much as needed for your game:

Code Listing 24: Race class

```
public class Race
{
    public string Name { get; set; }
    public string Description { get; set; }

    private Dictionary<string, int> statModifiers;

    private List<Modifier> weaknesses;
    private List<Modifier> resistances;

    public Race()
    {

    }

    public Race(string name)
    {
        Name = name;
    }
}
```

The stat modifiers are simply a positive or negative value, so we'll use a **`Dictionary`** to hold the values. The stat abbreviation, such as **`INT`**, is the key.

In addition to being better at some things than other races, a member of a race might have some disadvantages that we'll need to be able to identify. Both advantages and disadvantages can be tracked using the same class, the **`Modifier`** class:

Code Listing 25: Modifier class

```
public enum ModifierType
{
    Fire,
    Water,
    Magic,
    Disease,
    Poison
}

public class Modifier
```

```
{
    public ModifierType Type;
    public int Amount;
}
```

A couple of things a character can be affected by are noted in the **`ModifierType`** enum. Feel free to expand on this as needed for your game.

For a weakness, the **`Amount`** property would have a negative value. A resistance would set the **`Amount`** to a positive value. This allows you to have the same calculation handle both situations.

Classes

You can think of a character's class as their job or profession. A class usually gives you a good idea of what the character is good at without knowing anything else about them. Some sample classes are:

- **Fighter**: Physically tough combat specialist, usually in melee combat.
- **Mage**: Intelligent magic user who is usually weaker physically.
- **Thief**: Nimble, dexterous person who can get into places others can't.
- **Cleric**: Powered by their god, a combination fighter/mage usually, although sometimes just a mage.
- **Monk**: Usually another combination fighter/mage where their "magic" comes from themselves. Often a hand-to-hand specialist, but sometimes uses a couple specific types of weapons like staves or maces.
- **Ranger**: A fighter type who is good at outdoor skills such as tracking and hiding, and who usually specializes in using a bow.

Every class usually is a trade-off where the character will be better at some things, and not so good at others. We'll see how this works later on when we look at skills.

The **`EntityClass`** will hold the specifics of a class a character can select. We have to use this name as the word "class" is reserved in C#.

Code Listing 26: EntityClass (profession) class

```
public class EntityClass
{
    public string Name = "";
    public string Description = "";
    public DieType HPDice;

    private Dictionary<string, int> statModifiers;

    public int StatModifiersCount
    {
        get { return (statModifiers != null) ? statModifiers.Count : 0; }
    }
```

```
    public EntityClass()
    {

    }

    public EntityClass(string name)
    {
        Name = name;
    }
}
```

There are some functions in the class for dealing with the **`statModifiers`** that aren't included here. They're very straightforward, but feel free to review them in the project code.

The **`HPDice`** uses the **`DieType`** enum to specify how the hit points for a class are calculated. The hit points value is a number that represents the amount of damage a character can sustain before being killed. Usually this will increase over time as a character gains experience from things like doing quests or killing things.

The **`statModifiers`** use the stat abbreviation as the key, and a positive or negative number as the value.

Class equipment

Often, a class will automatically give a character equipment to start off with, such as armor and weapons. This allows the player to get started actually playing the game quicker. Alternatively, you can allow the player to choose the character's equipment from a list of items that are allowed to be used by the character's class. This lets the player personalize the character a bit to fit how they want to play but will take more time to implement.

We'll cover equipment in a later chapter and show a number of ways to restrict items.

The entity

We have all the information that's needed for our character at this point—we just need something to hold all of it. We'll actually use two classes: a base class for all entities, and one specific to a player character.

The **`Entity`** class holds all of the information we've discussed so far and will be used for both the player character and non-player characters in the game. We'll talk about the latter in the next chapter.

Code Listing 27: Entity class

```
[Serializable]
```

```
public class Entity
{
    public EntityType Type { get; set; }

    public string Name { get; set; }
    public string ClassID { get; set; }
    public byte Level { get; set; }

    public string RaceID { get; set; }

    public short BaseHP { get; set; }
    private short curHP;

    public EntityAlignment Alignment { get; set; }

    public EntitySex Sex { get; set; }
    public short Age { get; set; }

    private List<EntityStat> stats;

    public string PortraitFileName { get; set; }

    public string SpriteFilename { get; set; }

    //Requires a strength stat
    private float maxWeight;

    private List<string> knownNPCs;


    public void AddKnownNPC(string name)
    {
        if (knownNPCs == null)
            knownNPCs = new List<string>();

        knownNPCs.Add(name);
    }

    public bool CheckKnownNPC(string name)
    {
        if (knownNPCs == null)
            return false;

        return knownNPCs.Contains(name);
    }

    public int MaxWeight()
    {
        //Find the strength stat
```

```
        return stats.Find(s => s.StatName == "strength").CurrentValue &
Stat.PoundsPerStatPoint;
    }

    public void AddStat(EntityStat stat)
    {
        if(stats == null)
        {
            stats = new List<EntityStat>();
        }

        stats.Add(stat);
    }
}
```

The class is marked **`Serializable`** in order to allow us to easily save entities to a file.

The **`Type`** member will allow us to distinguish between the different types of entities we'll have in our game:

Code Listing 28: EntityType enum

```
public enum EntityType
{
    Character,
    NPC,
    Creature,
    Monster
}
```

`Character` and **`NPC`** are probably pretty clear; the differences between **`Creature`** and **`Monster`** are more subtle. A creature can be an animal or something that isn't fantastic. A monster is the typical supernatural type of thing you fight in D&D—anything from an ogre to a dragon to a zombie.

The **`ClassID`** is the name of the **`EntityClass`** object that identifies the RPG class of the entity.

`Level` is the numeric value that is representative of the experience the entity has. For something other than the character, this will allow the game to have multiple entities of the same type that vary in difficulty to defeat. We're limiting the value to the max value of a **`byte`** data type, which is 255. If you want to have entities that are higher than that, simply change it to a **`short`**.

`RaceID` is the name of the race of the entity. This will only be applicable for humanoid-type entities: the **`Character`** and **`NPC`**.

Since the entity can be damaged and healed during the game, we need to track both the current amount of hit points the entity has and the maximum hit points the entity can have. The **curHP** member is private since we need to ensure its value doesn't get set above the **MaxHP**; when the value is 0 or less, we need to handle having the entity die.

The **Alignment** member is something you may not want to use in your game. In many pencil-and-paper RPG systems, the alignment of an entity is used to determine how the entity acts. This could be something as simple as what we'll use, to something as complex as the nine different types of alignment in D&D.

Here's what we'll have:

Code Listing 29: EntityAlignment enum

```
public enum EntityAlignment
{
    Good,
    Neutral,
    Evil
}
```

In a video game RPG, having an alignment allows you to have items that can only be used by a certain alignment, spells that affect a certain alignment, and tailor quests based on an alignment.

The **PortraitFileName** allows you to specify a graphic that can be displayed in the game's UI to represent the character. The Neverwinter Nights games do this:

Figure 2: Character portrait example

This allows your game to quickly show the player information about the entities in the game—how much HP the entity has, for example. You can also use this in your combat system to show the order in which entities act.

The **SpriteFileName** will be used in conjunction with the graphics system from the previous chapter. This allows us to keep game-system-specific code separate from the RPG system code.

If you want to prohibit the character from carrying tons of loot and gear, the **MaxWeight** member will let you do this. The simple calculation can be changed, or you can tweak the multiplier constant to your liking.

The character

There is some information that will be used only by the character object, so we have a class that inherits from the **Entity** class:

Code Listing 30: Character class

```
[Serializable]
public class Character : Entity
{
    public int Experience { get; set; }

    // Other class member fields will go here

}
```

Currently, we just have a member that tracks the experience of the character. Non-character entities don't usually increase in power or abilities, so just having the **Level** member is fine. We'll add to this class in later chapters.

You can take a look at this chapter's project to see how we put all these classes together to allow the player to create their character. You'll also see code for how we implemented some basic UI elements to allow the player to make their selections. By necessity, it's very simple and not as elegant as it could be. The UI for your game is something that you can slowly upgrade as you have time. Getting a version of your game up and running quickly will be important for getting a feel of the gameplay and allowing you to start balancing the game to make sure it's fun and fair for the player.

What's next

Now that we have a character for the player, and that character can be moved around the game world, we need to make that world a bit more interactive. We'll start in the next chapter by allowing the character to talk to the entities that inhabit that world. We'll do this by implementing a simple, but somewhat versatile conversation system. This will be the first step in allowing the player character to take on the quests that are typical for RPGs.

Chapter 4 Conversations

Introduction

Your game is going to be very boring if there aren't entities for the player to interact with, both in an aggressive and a non-aggressive manner. This means we'll have to provide both types somehow, as well as handle how to interact with them. This chapter will show you how to do that.

We laid the foundation for having non-player characters in our game in the last chapter. We'll begin the process of adding them to our sample game in this one.

Conversation system

The first interaction a character usually has with a non-aggressive humanoid NPC is a conversation. Given that we don't yet have the ability to put in AI that will pass the Turing test, the player's conversations with NPCs will be somewhat limited, but how limited is up to you. A conversation can have many branches from the initial interaction, even looping and crossing each other. It could also be short and limited to just a couple of responses.

Our conversation system will give you the ability to have something in the middle. The conversations you create can be as long as you want, with as many choices as you want (at least, within what will fit on the screen). It will be up to you to make them interesting.

Code-wise, a conversation is a hierarchy of objects that have, at a minimum, text to display to the player and a number of objects that the player can choose as a response if one is needed. These responses can potentially have responses for the NPC and so on, letting the player interact with the NPC to whatever end you want. A conversation can give the player information that is needed to advance the game, allow the player to receive a question from an NPC, buy or sell items, and so on.

If you diagrammed a conversation, it could look something like this:

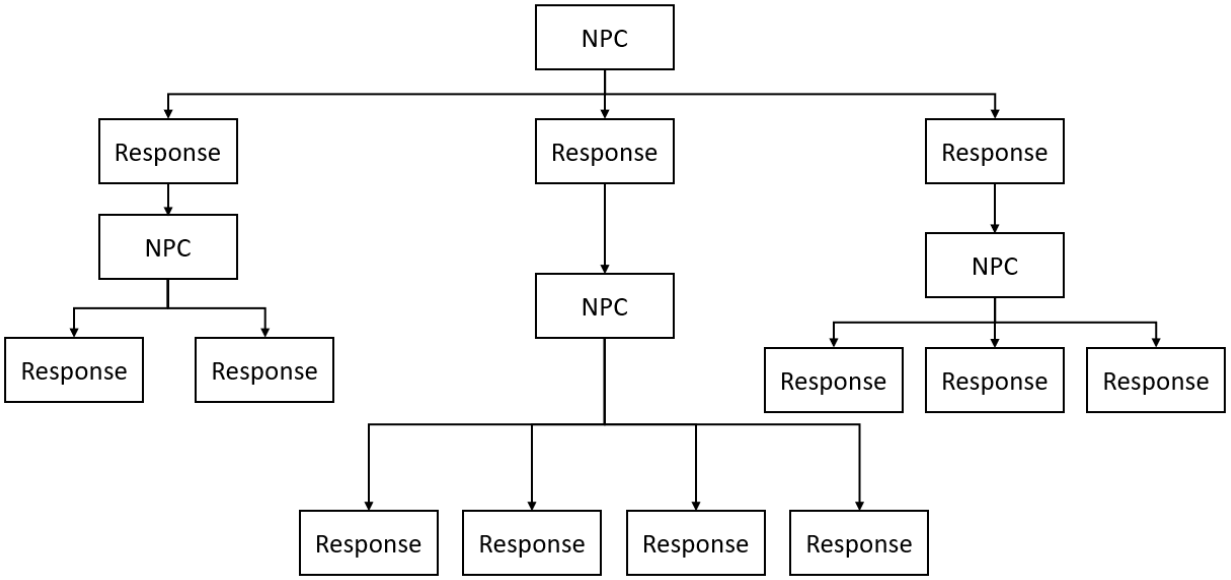


Figure 3: Conversation diagram

The number of responses, while usually consistent throughout a conversation, could vary. Realistically, you'll probably never need more than a half dozen, although the usual amount is three or four. Some responses might also be dynamic based on conditions in the game, actions taken by the player, statistics of the character, and so on. The new Cyberpunk 2077 game has responses based on the background the player selected for their character. An NPC might offer completely different responses if the character has just finished a rampage through the town, for example, or even refuse to speak with the character.

Pre-function

The ability to make an evaluation before a node in the conversation is shown will be added. This will be a simple true/false check but will still provide a good bit of flexibility. For example, if the player attempts to speak to an NPC who has a quest for them, we might want to check to see if the player already has that quest assigned to them to show a proper greeting. If the player doesn't have the quest, we would proceed as normal. If the player does have the quest, the NPC might want to ask if they have completed it.

If we were to diagram this, it would look like the following:

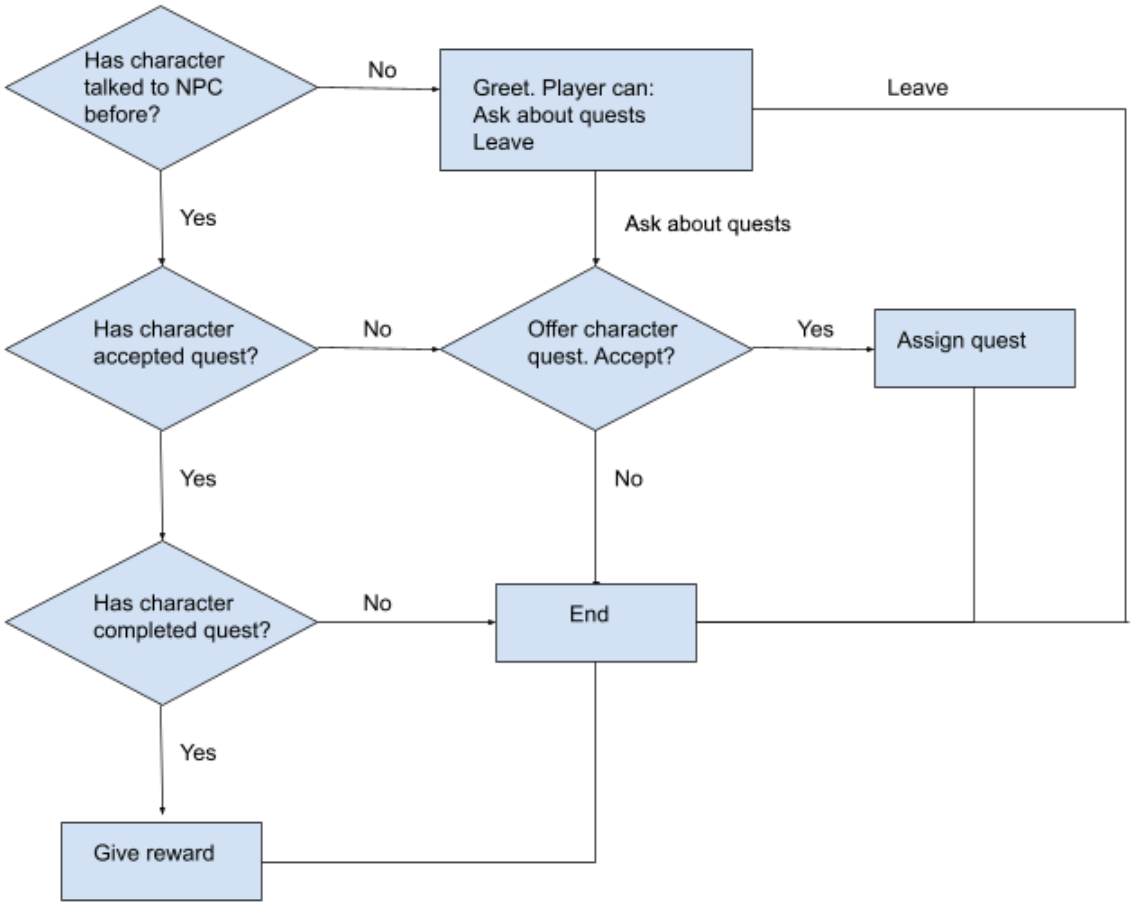


Figure 4: Conversation flow diagram

Post-function

Like the pre-function check, having the ability to do something after the player has selected a response is very useful. In our sample, if the player chooses to accept the quest the NPC offers, we need to add that quest to their character.

Let's take a look at how our sample dialog integrates both of these features. The data we'll use for this conversation will be contained in a JSON file:

Code Listing 31: Conversation JSON data

```
{
  "id": 1,
  "nodes": [
    {
      "id": 1,
```

```
      "text": "Hello.",
      "nodeFunctionType": 1,
      "functionName": "CheckKnownNPC",
      "functionParams": [ "1" ],
      "nodeCaseType": 0,
      "responses": [
        {
          "id": 2,
          "text": "",
          "nodeFunctionType": 0,
          "functionName": null,
          "functionParams": null,
          "nodeCaseType": 1,
          "responses": [
            {
              "id": 4,
              "text": "Hello",
              "nodeFunctionType": 1,
              "functionName": "QuestAssigned",
              "functionParams": [ "1" ],
              "nodeCaseType": 0,
              "responses": [
                {
                  "id": 7,
                  "text": "",
                  "nodeFunctionType": 0,
                  "functionName": null,
                  "functionParams": null,
                  "nodeCaseType": 1,
                  "responses": [
                    {
                      "id": 11,
                      "text": "I've finished the quest.",
                      "nodeFunctionType": 2,
                      "functionName": "IsQuestCompleted",
                      "functionParams": [ "1" ],
                      "nodeCaseType": 0,
                      "responses": [
                        {
                          "id": 2,
                          "text": "",
                          "nodeFunctionType": 0,
                          "functionName": null,
                          "functionParams": null,
                          "nodeCaseType": 2,
                          "responses": [
                            {
                              "id": 9,
```

```
                                "text": "You've not done what I asked. Come
back when you're finished.",
                                "nodeFunctionType": 0,
                                "functionName": null,
                                "functionParams": null,
                                "nodeCaseType": 0,
                                "responses": [
                                  {
                                    "id": 15,
                                    "text": "Good bye.",
                                    "nodeFunctionType": 0,
                                    "functionName": null,
                                    "functionParams": null,
                                    "nodeCaseType": 0,
                                    "responses": null
                                  }
                                ]
                              }
                            ]
                          },
                          {
                            "id": 12,
                            "text": "",
                            "nodeFunctionType": 0,
                            "functionName": null,
                            "functionParams": null,
                            "nodeCaseType": 1,
                            "responses": [
                              {
                                "id": 11,
                                "text": "Very good. Here is your reward.",
                                "nodeFunctionType": 2,
                                "functionName": "CompleteQuest",
                                "functionParams": [ "1" ],
                                "nodeCaseType": 1,
                                "responses": [
                                  {
                                    "id": 15,
                                    "text": "Good bye.",
                                    "nodeFunctionType": 0,
                                    "functionName": null,
                                    "functionParams": null,
                                    "nodeCaseType": 0,
                                    "responses": null
                                  }
                                ]
                              }
                            ]
                          }
```

```
                    ]
                  }
                ]
              },
              {
                "id": 8,
                "text": "",
                "nodeFunctionType": 0,
                "functionName": null,
                "functionParams": null,
                "nodeCaseType": 2,
                "responses": [
                  {
                    "id": 13,
                    "text": "Good bye.",
                    "nodeFunctionType": 0,
                    "functionName": null,
                    "functionParams": null,
                    "nodeCaseType": 0,
                    "responses": null
                  }
                ]
              }
            ]
          }
        ]
      },
      {
        "id": 3,
        "text": "",
        "nodeFunctionType": 0,
        "functionName": null,
        "functionParams": null,
        "nodeCaseType": 2,
        "responses": [
          {
            "id": 5,
            "text": "Do you have a quest for me?",
            "nodeFunctionType": 2,
            "functionName": "HasQuest",
            "functionParams": null,
            "nodeCaseType": 0,
            "responses": [
              {
                "id": 3,
                "text": "",
                "nodeFunctionType": 0,
                "functionName": null,
                "functionParams": null,
```

```
                  "nodeCaseType": 1,
                  "responses": [
                    {
                      "id": 9,
                      "text": "Yes. I need you to kill all the goblins
outside of town.",
                      "nodeFunctionType": 0,
                      "functionName": null,
                      "functionParams": null,
                      "nodeCaseType": 1,
                      "responses": [
                        {
                          "id": 14,
                          "text": "I don't have time for that.",
                          "nodeFunctionType": 0,
                          "functionName": null,
                          "functionParams": null,
                          "nodeCaseType": 0,
                          "responses": null
                        },
                        {
                          "id": 15,
                          "text": "Sure, I'd be glad to.",
                          "nodeFunctionType": 2,
                          "functionName": "AssignQuest",
                          "functionParams": [ "1" ],
                          "nodeCaseType": 0,
                          "responses": null
                        }
                      ]
                    }
                  ]
                },
                {
                  "id": 3,
                  "text": "",
                  "nodeFunctionType": 0,
                  "functionName": null,
                  "functionParams": null,
                  "nodeCaseType": 2,
                  "responses": [
                    {
                      "id": 10,
                      "text": "No, I'm sorry.",
                      "nodeFunctionType": 0,
                      "functionName": null,
                      "functionParams": null,
                      "nodeCaseType": 2,
                      "responses": [
```

```
                    {
                      "id": 16,
                      "text": "Good bye.",
                      "nodeFunctionType": 0,
                      "functionName": null,
                      "functionParams": null,
                      "nodeCaseType": 0,
                      "responses": null
                    }
                  ]
                }
              ]
            }
          ]
        },
        {
          "id": 16,
          "text": "Good bye.",
          "nodeFunctionType": 0,
          "functionName": null,
          "functionParams": null,
          "nodeCaseType": 0,
          "responses": null
        }
      ]
    }
  ]
}
```

The first node contains a pre-function that has the conversation system check the player character to see if it has previously interacted with the entity it is having the conversation with. Since the conversation system knows about the entity that is having the conversation, as we'll see shortly, there's no need to have that information in the conversation file. The two responses for this node are **`true`** and **`false`**, which every node that contains a pre-function must have in order for the function to work.

If the player has interacted with the entity, another pre-function checks to see if the player already has the quest that the entity can offer. The **`functionParams`** property of the node holds the ID of the quest. This property allows for any number of pieces of data; they would be comma-delimited between the brackets.

If the quest is already assigned, a check is made to see if the player has completed it. We'll see exactly how this is accomplished when we look at our quest system in the next chapter. If it hasn't been completed, the player is told to come back when it has been; otherwise the quest reward is given.

If you look through the JSON, you'll see another piece of the system that we'll discuss: a placeholder for the quest description in node 9. Since a conversation could potentially allow multiple quests to be offered, this information needs to be determined when the conversation is occurring. There could be a number of pieces of information like this, such as the name of an entity or a location the player needs to find, or a number of items or entities the player needs to obtain or eliminate.

The code to load a conversation is just a few lines:

Code Listing 32: Method to load a conversation

```
public static Conversation LoadConversation(string id)
{
    Conversation conversation = new Conversation();

    string data = File.ReadAllText(@"Content\Data\Conversations\" +
id.ToString() + ".json");

    conversation = JsonConvert.DeserializeObject<Conversation>(data);

    return conversation;
}
```

The Newtonsoft library that we're using handles all the heavy lifting in one line: the **`JsonConvert.DeserializeObject`** method.

Now that we've looked at the conversation, we'll go over the code that allows it to work.

There are three main classes for this system: **`Conversation`**, **`ConversationManager`**, and **`ConversationRenderer`**.

The **`ConversationManager`** is extremely simple at this point. It has only five members:

Code Listing 33: ConversationManager class

```
public class ConversationManager
{
    Conversation conversation;
    Entity player;
    Entity npc;

    ConversationNode curNode;

    public bool IsActive;
}
```

The constructor for the class is passed the first three members. **`IsActive`** lets other systems know if a conversation is currently taking place. **`curNode`** provides the information for the line to be displayed based on what the entity being talked to is saying. This is the data in a node of the JSON file.

Code Listing 34: ConversationNode class

```
public enum CaseType
{
    CaseNone,
    CaseTrue,
    CaseFalse
}

public enum FunctionType
{
    FunctionNone,
    PreFunction,
    PostFunction
}

public class ConversationNode
{
    public int ID;
    public string Text;
    public FunctionType NodeFunctionType;
    public string FunctionName;
    public string[] FunctionParams;
    public CaseType NodeCaseType;

    private List<ConversationNode> responses;
}
```

Nothing here is that surprising. The class is simply a holder for data.

The **`Conversation`** class is just about as simple:

Code Listing 35: Conversation class

```
public class Conversation
{
    private int id;
    public List<ConversationNode> nodes;
    private ConversationNode curNode;
    public ConversationStatus Status;
}
```

The only complex piece in the class is the code that deals with the pre-function:

Code Listing 36: Method to check a node before it displays

```
public void CheckPreFunction()
{
    if (curNode.NodeFunctionType == FunctionType.PreFunction)
    {
        int index = 0;

        object obj =
Globals.FunctionClasses[(ConversationFunctions)Enum.Parse(typeof(Conversati
onFunctions), curNode.FunctionName)];

        if (obj.GetType().IsGenericType && obj is IList)
        {
            index = Convert.ToInt32(curNode.FunctionParams[0]);

            //First function param would be the id of the object in the
list
            object ret =
Globals.FunctionClasses[(ConversationFunctions)Enum.Parse(typeof(Conversati
onFunctions),
curNode.FunctionName)].GetType().GetMethod(curNode.FunctionName)
                .Invoke(((IList)obj)[index], new[] { curNode.FunctionParams
});

            if ((bool)ret)
            {
                curNode.Responses = curNode.Responses.Find(c => c.Text ==
"true").Responses;
            }
            else
            {
                curNode.Responses = curNode.Responses.Find(c => c.Text ==
"false").Responses;
            }
        }
        else
        {
            object ret =
Globals.FunctionClasses[(ConversationFunctions)Enum.Parse(typeof(Conversati
onFunctions),
curNode.FunctionName)].GetType().GetMethod(curNode.FunctionName)
                .Invoke(obj, new[] { curNode.FunctionParams });

            if ((bool)ret)
            {
                curNode.Responses = curNode.Responses.Find(c =>
c.NodeCaseType == CaseType.CaseTrue).Responses;
            }
```

```
            else
            {
                curNode.Responses = curNode.Responses.Find(c =>
c.NodeCaseType == CaseType.CaseFalse).Responses;
            }
        }
    }
}
```

Code Listing 37: Method to process a response selection

```
public void SelectResponse(int index)
{
    //Check post function
    if (curNode.Responses[index].NodeFunctionType ==
FunctionType.PostFunction && curNode.Responses != null)
    {
        int param = 0;

        object obj =
Globals.FunctionClasses[(ConversationFunctions)Enum.Parse(typeof(Conversati
onFunctions), curNode.Responses[index].FunctionName)];
        object ret;

        if (obj.GetType().IsGenericType && obj is IList)
        {
            //First function param would be the id of the object in the
list
            param =
Convert.ToInt32(curNode.Responses[index].FunctionParams[0]);

            ret =
Globals.FunctionClasses[(ConversationFunctions)Enum.Parse(typeof(Conversati
onFunctions),
curNode.Responses[index].FunctionName)].GetType().GetMethod(curNode.Respons
es[index].FunctionName)
                .Invoke(((IList)obj)[param], new[] {
curNode.Responses[index].FunctionParams });
        }
        else
        {
            string[] functionParams =
curNode.Responses[index].FunctionParams;

            ret =
Globals.FunctionClasses[(ConversationFunctions)Enum.Parse(typeof(Conversati
```

```
onFunctions),
curNode.Responses[index].FunctionName)].GetType().GetMethod(curNode.Respons
es[index].FunctionName)
                .Invoke(obj, functionParams);
        }

        if (curNode.Responses != null)
        {
            if ((bool)ret)
            {
                curNode = curNode.Responses[index].Responses.Find(c =>
c.NodeCaseType == CaseType.CaseTrue);
            }
            else
            {
                curNode = curNode.Responses[index].Responses.Find(c =>
c.NodeCaseType == CaseType.CaseFalse);
            }
        }
        else
        {
            curNode = null;
        }
    }
    else
    {
        if (curNode.Responses[index].Responses == null)
        {
            //End conversation
            curNode = null;
        }
        else
        {
            curNode = curNode.Responses[index];
        }
    }

    if (curNode == null)
    {
        Status = ConversationStatus.Completed;
    }
}
```

As with the **`CheckPreFunction`** method, the only complex piece here is calling the post-function method if one exists, but it's the exact same code as we've seen.

If the current node has no responses, we end the conversation.

ConversationRenderer

Drawing the conversation interface involves drawing a background window in which the conversation text is displayed, as well as the current node's text and responses:

Code Listing 38: ConversationRenderer class

```
// In gameplay screen
if (conversationManager != null && conversationManager.IsActive)
    conversationRenderer.Render(spriteBatch,
conversationManager.GetCurrentNode());

public class ConversationRenderer
{
    private Texture2D background;
    private Rectangle rect;
    private Vector2 conversationLine;

    private SpriteFont font;

    public ConversationRenderer()
    {
        ContentManager content = Game1.GetContentManager();

        background =
content.Load<Texture2D>("Sprites/UI/conversationbackground");
        rect = new Rectangle(0,
Game1.GetScreenManager().GraphicsDevice.Viewport.Height - 100,
Game1.GetScreenManager().GraphicsDevice.Viewport.Width, 100);
        font = content.Load<SpriteFont>("conversationfont");

        conversationLine = new Vector2(rect.X + 15, rect.Y + 5);
    }

    public void Render(SpriteBatch spriteBatch, ConversationNode curNode)
    {
        if (curNode != null)
        {
            spriteBatch.Draw(background, rect, Color.White);

            spriteBatch.DrawString(font, curNode.Text, conversationLine,
Color.Black);

            int y = (int)conversationLine.Y + 15;
            int x = (int)conversationLine.X + 20;

            int i = 0;

            if (curNode.Responses != null)
```

```
            {
                foreach (ConversationNode node in curNode.Responses)
                {
                    spriteBatch.DrawString(font, (i + 1).ToString() + ")  "
+ node.Text, new Vector2(x, y + (20 * i)), Color.Black);
                    i++;
                }
            }
            else
            {
                spriteBatch.DrawString(font, (i + 1).ToString() + ")  Leave
conversation.", new Vector2(x, y + (20 * i)), Color.Black);
            }
        }
    }
}
```

NPCs

Now that we have a conversation, we just need to have an NPC to talk to. A full game will have a lot of NPCs and other entities. Every area in the game world will have entities: good, bad, and neutral. When an area is loaded, it will have to load the entities that populate it. An easy way to do this is one that we already know: store the data for them in a JSON file. Here's the one we'll use for our small demo:

Code Listing 39: NPC JSON data

```
{
  "$type":
"System.Collections.Generic.List`1[[MonoGameRPG.EntityGameObject, Chapter
4]], mscorlib",
  "$values": [
    {
      "$type": "MonoGameRPG.EntityGameObject, Chapter 5",
      "EntityType": 1,
      "Entity": {
        "$type": "RPGEngine.Entity, Chapter 5",
        "ID":  1,
        "Type": 0,
        "Name": "Blacksmith",
        "ClassID": "1",
        "Level": 1,
        "RaceID": "1",
        "BaseHP": 10,
        "Alignment": 0,
        "Sex": 0,
        "Age": 0,
```

```
        "PortraitFileName": null
      },
      "Target": null,
      "GameSpriteFileName": "Sprites/Test/TestSheet2",
      "Type": 1,
      "StartLocation": "2, 2"
    },
    {
      "$type": "MonoGameRPG.EntityGameObject, Chapter 4",
      "EntityType": 1,
      "Entity": {
        "$type": "RPGEngine.Entity, Chapter 4",
        "ID": 2,
        "Type": 0,
        "Name": "Shopkeeper",
        "ClassID": "1",
        "Level": 1,
        "RaceID": "1",
        "BaseHP": 10,
        "Alignment": 0,
        "Sex": 0,
        "Age": 0,
        "PortraitFileName": null
      },
      "Target": null,
      "GameSpriteFileName": "Sprites/Test/TestSheet3",
      "Type": 1,
      "StartLocation": "6, 6"
    }
  ]
}
```

The code to load this is not much different than what we've already used. The only difference is that we need to add a setting that tells the **`DeserializeObject`** method how to handle the **`EntityGameObject`**:

Code Listing 40: JSON deserializtion settings

```
JsonSerializerSettings settings = new JsonSerializerSettings
{
    TypeNameHandling = TypeNameHandling.All
};
```

Pass this as the second parameter to the method, and everything just works.

Once we have the data loaded, we can create our NPCs. The loading code is in a method called **`LoadNPCs`**; we just return the list that's created. We'll also add a hook to allow us to start a conversation when the entity is clicked:

Code Listing 41: Entity initialization to handle conversations

```
List<EntityGameObject> npcs;

npcs = new List<EntityGameObject>();
npcs.AddRange(GameObject.LoadNPCs());

foreach (EntityGameObject obj in npcs)
{
    obj.Initialize(ScreenManager.Game, obj.GameSpriteFileName);
    obj.NPCClicked += NPCClicked;
}

((Entity)npcs[0].Entity).AddConversation(ConversationManager.LoadConversati
on("1"));
((Entity)npcs[0].Entity).AddQuest(1);
```

Although we're going to look at quests later, we'll add a placeholder to the first entity to allow the conversation system to work.

The gameplay code looks for input every frame and passes that input to every object to see if it was interacted with. This includes the entities, which allows us to start the conversation when one is clicked:

Code Listing 42: Method for handling NPC being clicked

```
private void NPCClicked(NPCClickedEventArgs e)
{
    Entity entity = (Entity)npcs.Find(npc => ((Entity)npc.Entity).ID ==
e.ID).Entity;

    if (conversationManager == null)
    {
        conversationManager = new
ConversationManager(((Entity)npcs.Find(npc => ((Entity)npc.Entity).ID ==
e.ID).Entity).GetConversation(e.ConversationID),
((Entity)character.Entity), entity);
    }

    conversationManager.Start();
    conversationManager.IsActive = true;
}
```

The end result of all this is a simple interface that most RPG players will be familiar with:

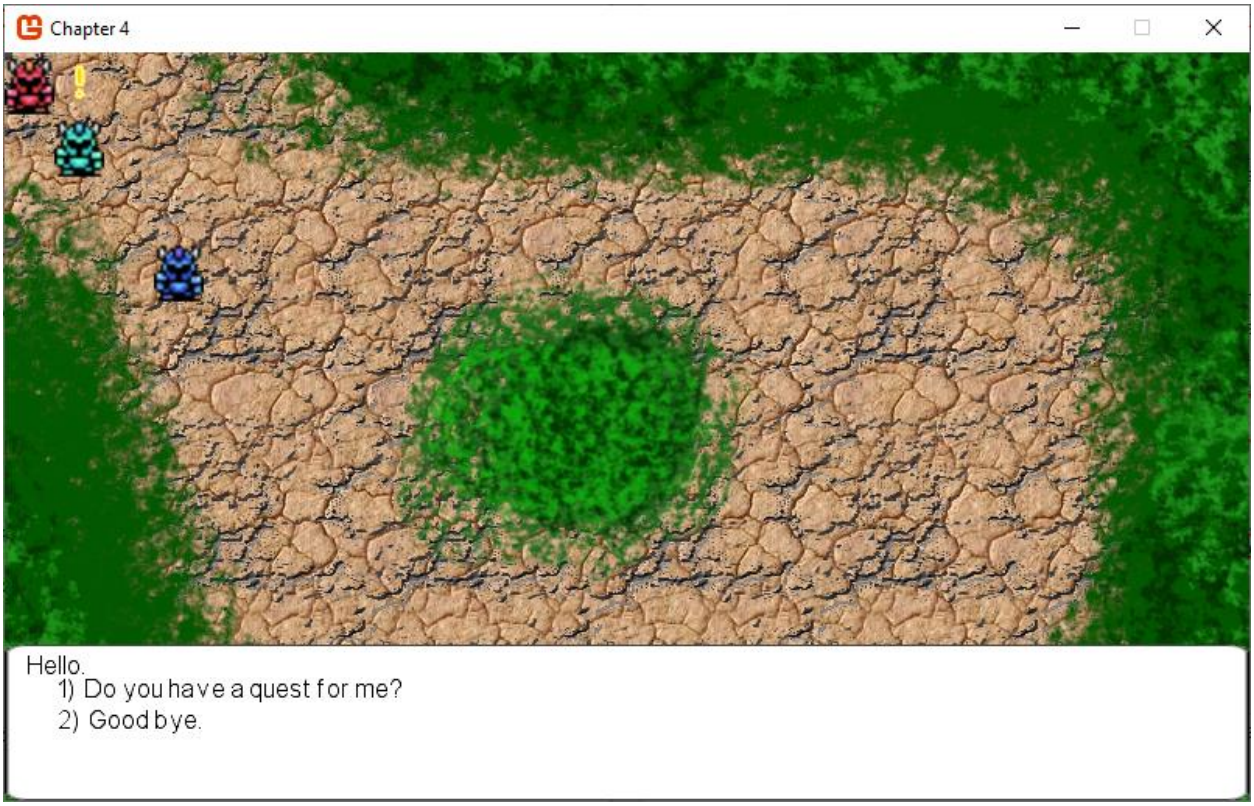


Figure 5: Conversation in action

As this is just a high-level view of how the conversation system works, we encourage you to run the sample for this chapter to see the conversation system in action and take a look at all the relevant code involved as there is a bit more to it.

We've looked at the most important pieces, and this should be enough to give you an idea of how to implement a conversation system. We'll expand on this to tie into our quest system and give the player a goal to accomplish.

What's next

Now that we have the ability to have the player accept a quest, we'll work on actually designing the quest system. Head over the the next chapter where we'll take a look at a simple but relatively robust quest system we'll implement.

Chapter 5 Quests

Introduction

Quests are a major part of most RPGs. They give the character purpose and direction. Without a quest, the character could just wander around the world and potentially never accomplish anything worthwhile. The story of the game would never go anywhere without quests to drive it forward. The quests of the game are basically the plot of the story.

Quests can be extremely simple or as complex as you want them to be. It all depends on your game and how deep you want the world and story to be. Someone telling the character to go somewhere and kill something and provide proof it's dead is pretty simple—basically three steps. It would also probably be a pretty boring game if it were just those three steps. Usually there's steps in between, and sometimes entire quests. These are sub-quests, which make the game a bit more complicated and hopefully more interesting.

In our example, if the thing to be killed can only be killed by a specific magic weapon, the player may have to find that weapon first. This could lead the player to a person who has the weapon who may task the player with doing something for them before giving the player the weapon. This would be a sub-quest and could potentially lead to more sub-quests. It all depends on how difficult you want to make it for the player.

In constructing your quests, you'll want to make sure that the steps are meaningful and interesting. "Go here, do this," without giving the player a reason could end up boring the player and make them question the game, or even just stop playing it.

Storytelling is an art, and constructing quests is basically a form of storytelling. Like just about anything, you'll get better with practice and feedback. Listen to people that play and test your game, and learn what makes them interested and makes them have fun.

The sample quest we'll construct in this chapter is going to be the three-step quest we just talked about, but the framework we'll create for those three steps can easily be expanded upon, so feel free to do so once you've gone through the chapter.

The Quest classes

The **`Quest`** class is the start of the system, so let's take a look. Some members have been included that we're not going to use, but they will give you some ideas on how to expand the quest system.

Code Listing 43: Quest class

```
public class Quest
{
    public int ID;
    public string Name;
    public string Description;
    public QuestRewardType RewardType;
    public int RewardItemID;
    public List<int> AllowedClasses;
    public int RequiredLevel;
    public bool IsMultipleAllowed;

    // 0 means no limit
    public long TimeBetweenQuests;

    // Total time in which quest must be completed, 0 for none
    public long TimeLimit;

    // Is the reward known by the player when accepting the quest
    public bool IsRewardShown;

    // Array of QuestStep
    public List<QuestStep> Steps = new List<QuestStep>();
}
```

Of the members here, we're only going to use about half: the first five and the last one. Only two should not be immediately obvious: **`RewardType`** and **`RewardItemID`**. The first will currently be either money or an item. If an item, the **`RewardItemID`** will enable you to find it in your item system, which we'll look at in a later chapter. If the reward is money, this member will be the amount. We could have a separate member for the amount, but there's no reason not to reuse this one.

You can easily expand the type of reward to almost anything in your game, such as land, reputation, or an NPC to help the character out. The limit is only the time you want to take to implement them.

There's not much more going on in the **`QuestStep`** class than the **`Quest`** class. Again, we'll have some members you can use to expand your quests.

Code Listing 44: QuestStep class

```
public class QuestStep
{
    public string StepName;
    public int StepEntity;
    public InteractionType StepInteractionType;
    public QuestStepType Type;
    public string QuantityName;
    public int Quantity;

    // Array of IDs of sub-quests
    public List<int> SubQuests;

    // Required level to start the step, each step may have a different
level
    public byte MinimumLevel;

    // Number of minutes the step can take
    public long TimeLimit;

    public string JournalEntry;

    public bool Started;
}
```

Our quest will use two types of steps but a third is included.

Code Listing 45: QuestStepType enums

```
public enum QuestStepType
{
    FedEx,
    Item,
    Interact
}

public enum InteractionType
{
    Talk,
    Rescue,
    Kill,
    Get,
    Give
}
```

If the step involves the character interacting with another entity, the type of interaction is determined by the **`StepInteractionType`**. Our quest will involve two of these: **`Talk`** and **`Kill`**. The **`StepEntity`** member holds the ID of the entity with which the character will interact.

The data we'll need to track for quests the player has accepted will be a bit different than what the **Quest** class offers. We'll have a simple class for that:

Code Listing 46: AssignedQuest class

```
public class AssignedQuest
{
    public int QuestID;
    public int CurStep;
    public long TimeStepStarted;
    public long TimeStepFinished;
    public long TimeQuestStarted;
    public long TimeQuestFinished;
    public int QuestGiverID;
    //Number of items done for each step, if necessary. Key is object name.
    public Dictionary<string, int> NumItemsDone;
    public bool QuestFinished;
}
```

The two **TimeStep** members track when the current step was started, completed, and finished. This could be a bit confusing once the player has completed a step, as the **TimeStepFinished** will be for the previous step. You may never want to get to this level of detail, so feel free to delete them if you want. We tried to offer some flexibility in the quest system that some other quest systems offer.

The **NumItemsDone** requires a bit of explanation. The key in this dictionary is the name of the object. This could be an item, entity, or other kind of object, such as a crafting item. The value is the number of that object that has been collected. We're not using an ID since IDs aren't unique across all object types. There could be an item with an ID of **1** and an entity with an ID of **1**.

Now that we have our classes for holding quests, let's take a quick look at how they're assigned and tracked.

As we saw in the chapter on conversations, selecting a response that indicates a quest is to be assigned calls the method to do so. The actual code is fairly straightforward:

Code Listing 47: Method to handle assigning a quest

```
public bool AssignQuest(string id)
{
    if (AssignedQuests == null)
        AssignedQuests = new List<AssignedQuest>();

    AssignedQuest aq = new AssignedQuest();
    Quest q = QuestManager.LoadQuest(Convert.ToInt32(id));

    aq.QuestID = Convert.ToInt32(id);
    aq.TimeQuestStarted = aq.TimeStepStarted = DateTime.Now.ToBinary();
    aq.CurStep = 1;
```

```
    foreach(QuestStep step in q.Steps)
    {
        if (aq.NumItemsDone == null)
            aq.NumItemsDone = new Dictionary<string, int>();

        if (step.Quantity > 0)
        {
            aq.NumItemsDone.Add(step.QuantityName, 0);
        }
    }

    AssignedQuests.Add(aq);

    EventSystem.OnQuestAssigned(new EventSystem.EventSystemEventArgs() {
ObjectID = aq.QuestID });

    return true;
}
```

When the quest is assigned, we look through the steps to see if there are any that require the player to collect something. For each step that requires items, we add a placeholder to the dictionary.

After that's done, we call the **`OnQuestAssigned`** event to let other systems know the quest was assigned. For our demo, this just displays a message to let the player know the quest was assigned.

QuestManager and EventSystem classes

The **`QuestManager`** and **`EventSystem`** classes work together to handle all the details of coordinating events that happen that could be part of a quest. The events will be specific to every game, but there are some standard events that are usually part of a quest. For our sample, we're only using one: interacting with entities.

There are three steps to complete the quest we'll have, and all of them involve interacting with entities. Two of the steps are talking to an NPC to get and complete the quest, and one step is interacting with the goblins in the cave outside of town by killing them.

Code Listing 48: EventSystem class

```
public class EventSystem
{
    public class EventSystemEventArgs
    {
        public int ObjectID;    //Specific to event, Entity events = entity
ID, Item event = item ID, etc.
```

```
        public string Tag;      //Could be anything
    }

    public delegate void EventSystemEventHandler(EventSystemEventArgs e);

    public static event EventSystemEventHandler EntityKilled;
    public static event EventSystemEventHandler LootObtained;
    public static event EventSystemEventHandler LocationReached;
    public static event EventSystemEventHandler EntityTalkedTo;
    public static event EventSystemEventHandler ItemObtained;
    public static event EventSystemEventHandler LevelEntered;
    public static event EventSystemEventHandler QuestAssigned;
}
```

Code Listing 49: QuestManager class

```
public class QuestEventArgs
{
    public string Text;

    public QuestEventArgs(string text)
    {
        Text = text;
    }
}

public class QuestManager
{
    private Character character;

    public delegate void QuestEventHandler(QuestEventArgs e);

    public event QuestEventHandler QuestUpdated;

    public QuestManager(Character character)
    {
        EventSystem.EntityKilled += EventSystem_EntityKilled;
        EventSystem.EntityTalkedTo += EventSystem_EntityTalkedTo;
        EventSystem.ItemObtained += EventSystem_ItemObtained;
        EventSystem.LocationReached += EventSystem_LocationReached;
        EventSystem.LootObtained += EventSystem_LootObtained;
        EventSystem.LevelEntered += EventSystem_LevelEntered;

        this.character = character;
    }
```

```
    private void EventSystem_EntityTalkedTo(EventSystemEventArgs e)
    {
        //Quest type - Interact
        //InteractionType = Talk
        //StepEntity = e.ObjectID

        //Quest type - Fedex
        //GiveItemID = e.ObjectID

        foreach(Quest q in character.GetQuests())
        {
            AssignedQuest aq = character.AssignedQuests.Find(a => a.QuestID
== q.ID);
            if (aq != null)
            {
                QuestStep step = q.Steps[aq.CurStep];
                if (step.Type == QuestStepType.Interact
                && step.StepInteractionType == InteractionType.Kill
                && step.StepEntity == e.ObjectID)
                {
                    aq.CurStep++;
                    QuestUpdated(new QuestEventArgs(q.Name));
                }
            }
        }
    }

    private void EventSystem_EntityKilled(EventSystemEventArgs e)
    {
        //Quest type - Interact
        //InteractionType = Kill
        //StepEntity = e.ObjectID

        foreach (Quest q in character.GetQuests())
        {
            AssignedQuest aq = character.AssignedQuests.Find(a => a.QuestID
== q.ID);
            if (aq != null)
            {
                aq.NumItemsDone[e.Tag]++;

                QuestStep step = q.Steps[aq.CurStep];

                if (step.Type == QuestStepType.Interact
                    && step.StepInteractionType == InteractionType.Kill
                    && step.StepEntity == e.ObjectID
                    && step.Quantity == aq.NumItemsDone[e.Tag])
                {
                    aq.CurStep++;
```

```
                    QuestUpdated(new QuestEventArgs(q.Name));

                }
            }
        }
    }
}
```

The **EventSystem** class is kind of a go-between of the various systems that need to communicate with each other. The argument that's passed to the event uses an ID and a **Tag** to allow us to pass whatever kind of information we need. Usually this will be something like the ID of whatever object is involved in the event, an entity that's interacted with, or an item that's obtained.

The **Tag** is a string that gives us the flexibility to pass whatever kind of information that might be needed in addition to the object. The methods that are called for an event need to know what kind of data is in the argument, which should be obvious based on the event itself. An event that is used for interacting with an entity won't have an item ID.

The **QuestManager** class hooks into the **EventSystem** events in the constructor. It also saves a reference to the character.

When one of the events happens, the method that handles it looks through all the quests the character has been assigned. If it finds a matching quest, the conditions for advancing the quest are checked, which are dependent on the event. If the quest is advanced, an event is called to let other systems know.

We've added a handful of events though we'll only use a couple. Feel free to create your own quests that use all of the events.

Quest screen

The player will need to be able to see their current quests to know what they need to do next. There's a lot of info you can put on a quest screen, but we'll do the minimum to start: the steps, indicating which are completed, and the reward, if it can be shown. Here's a screenshot:

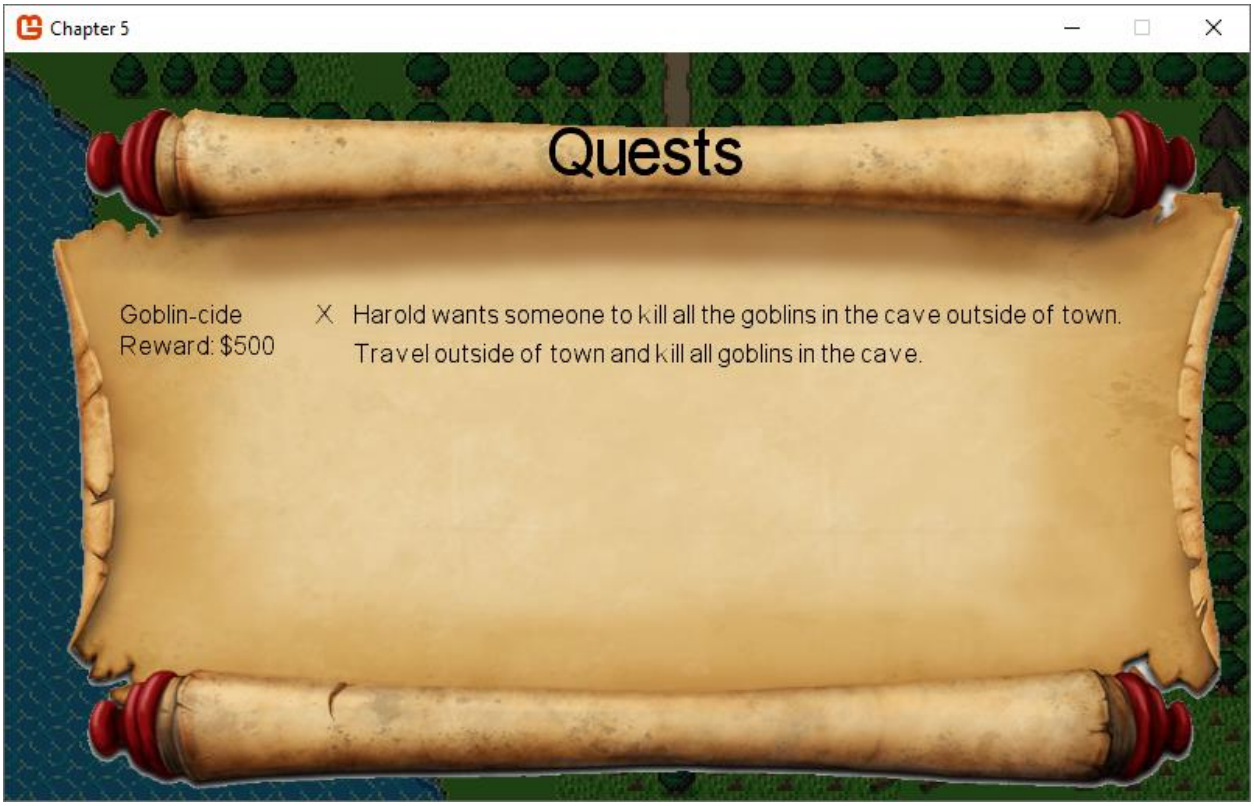


Figure 6: Quests screen

There's nothing really new here. We simply loop through the assigned quests, draw the name and the reward if the quest allows it, and loop through the steps, showing only the completed steps and the first uncompleted one.

We do have a "to do" item—since we don't have an item system yet, if the reward is an item, we just draw the item ID. We'll add that soon enough.

Completing quest steps

Completing quest steps basically involves monitoring input and passing that input along to all objects that need it. It's up to those objects to let the relevant systems know that something happened that could advance a quest. In our case, that means clicking an entity, either to kill a goblin or to interact with the NPC that gave the quest.

Outside of the conversation system, the **`GamePlayScreen`** class is notified by an entity when it's killed:

Code Listing 50: Method to handle an entity being killed

```
private void EntityKilled(EntityKilledEventArgs e)
{
```

```
    EntityGameObject entity = npcs.Find(n => ((Entity)n.Entity).ID ==
e.ID);

    //Add to character kill count

((Character)character.Entity).AddQuestItem(((Entity)entity.Entity).Name);

    //Remove entity from list so it's not rendered anymore
    npcs.Remove(entity);

    EventSystem.OnEntityKilled(new EventSystemEventArgs { ObjectID = e.ID,
Tag = ((Entity)entity.Entity).Name });
}
```

We keep track of what entities the character has killed since this is one of the possible items a character may need to collect. The event system is then notified, which invokes the corresponding event, letting any system that's monitoring it know. The **`QuestManager`** class is one of these systems, as we've seen.

Our quest is completed when the character interacts with the NPC in the town, letting him know the goblins have been killed. This is verified by checking the steps of the quest, making sure the quantity of goblins that needed to be killed have been.

If the quest is over, the player is given the reward and the quest is marked completed.

Enhancements

One thing that we've left for you to implement is a check of NPCs to see if the player has accepted a quest the NPC offers and to indicate this using a graphic. A graphic is part of the chapter source code. All that needs to be done is loop through the NPCs in the area of the character, see if they offer a quest and, if so, see if that quest ID is in the character's **`AssignedQuests`** list. If it is, display that icon.

What's next

We can now complete quests, although we've faked some parts, mainly the combat to kill the goblins. We'll take care of that shortly. We need to hook up the quests to NPCs and have the character be able to interact with them. For this, we'll need to have a game world for those NPCs to inhabit. We'll add the functionality to create that game world in the next chapter.

Chapter 6 Levels and Maps

In this chapter, we are going to look at how we can represent the area the player is currently in. This could be the interior of a shop, a town the player is wandering around in, or the deep, dark depths of a cold, damp dungeon.

We are going to do this with levels. Our level will have a number of maps that will help make up its detail. For the purposes of this book, we are keeping its bounds to the area of the screen. The level will have a number of maps passed to it. These will be textures that will represent the level floor layout and objects that are in the areas, as well as mobile objects (mobs), their patrol areas, and exits from the level to other areas of the game world.

We are also going to use our animated sprite sheet code that we covered in Chapter 2 when rendering the level.

Levels

For simplicity we have gone with a top-down view using hand-drawn maps, all rendered using a simple tile map system.

Our world is going to be made up of levels. The game world for the purpose of our book is going to be made up of the following areas:

Areas

Town

The town is where our intrepid adventurer is going to start. Within the town there will be the inn, a place where the player can buy some supplies and hear some rumors. The player will also be able to exit north through the town gates and enter the surrounding area.

Inn

The inn is full of ale, gossip, and rumors, one of which tells of a dungeon to the northeast that has become a home to a small party of goblins that have been raiding local farms. Our player can also buy some equipment here for their journey; a sword may be useful for those goblins.

Surrounding area

The village is not too far from the sea. Once leaving the village, the player will be able to roam the surrounding area freely, head to the dungeon that the rumors described, or even head back into town. Perhaps they will have a chance encounter while out in the wilds…

Dungeon

This is where the goblins are holed up, in the cellar of a keep that was once owned by a great wizard. After a frightful magical explosion a few hundred years ago, the only thing left of the keep is its cellar, located in the crater where the keep once stood.

Tile maps

Each level has a tile map to help describe where everything is in it. We pass a terrain map texture (this will be the ground tiles for the level), an overlay map (this will be for sprites that sit on top of the floor tiles), an object map (this will be used to map where items and other important objects are in the level), and finally, a map for mobs or NPCs (this will show where monsters and other NPCs may be in the level). Each pixel in the map is given a color to represent a sprite to be rendered at that location.

Sprite size

The first thing we need to do is decide how big the level tile maps are going to be. I have chosen to use 32×32 sprite sizes for areas of detail, so the town, the inn, and the dungeon. The surrounding area will be rendered using 16×16-pixel sprites, as we want this to seem like a more expansive area.

Map size

We are going to be rendering our sprites with 32×32 pixels, so we need to create our maps to fill the screen at that sprite size. The default screen size for our game is 800×480 pixels, so dividing both of those values by 32 tells us our maps for the 32×32 sprite size need to be 25×15 pixels.

Pretty much all our levels are going to be at this resolution, with the exception of the surrounding area level, which will use smaller sprites and create a larger playing area. For the area maps we are going with a 16×16 sprite, so all we need to do is double the map sizes we currently have, giving us tile map sizes of 50×30.

I am not going to cover all the maps. I will leave some of that up to you, but I will give examples of how we can create these level maps. Let’s take a look at the town maps we have created, and then we will look at the way our level base classes are made so we can use them to build any level we like by deriving from them and adding the specific code we want for that level.

Town maps

Terrain map

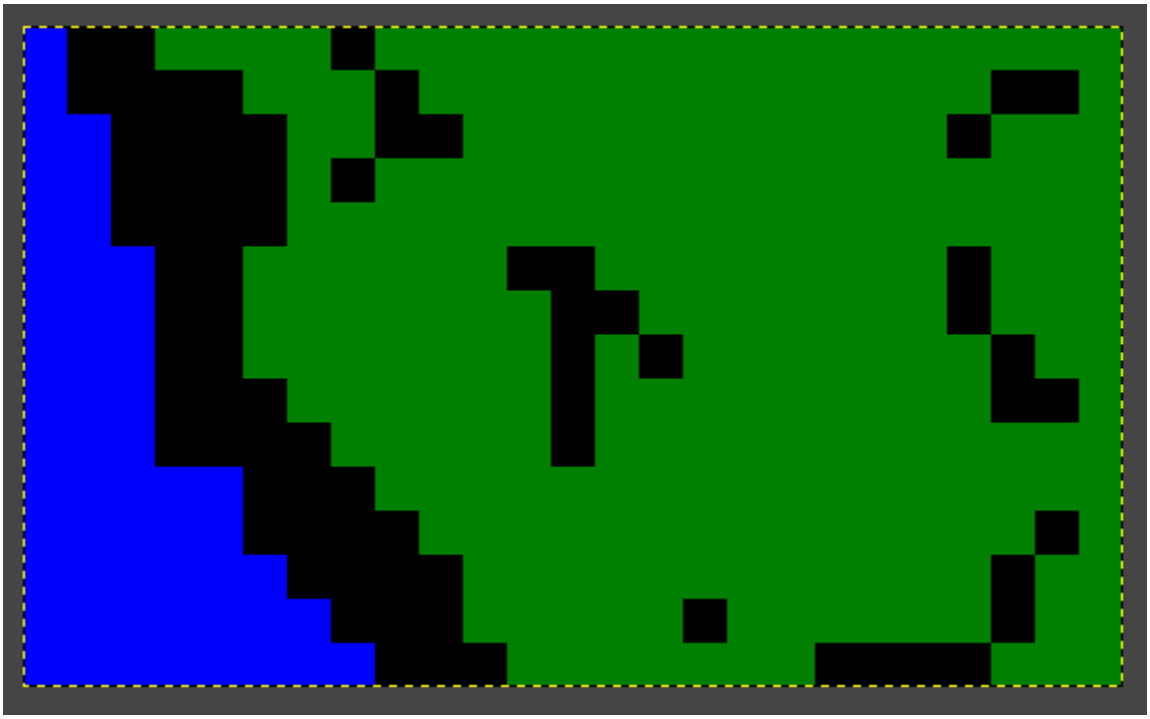


Figure 7: Town terrain map

This PNG texture created in GIMP 2.10.20 is 25×15 pixels in size. Each pixel is going to tell us what needs to be rendered at each sprite location on the screen. This is a very simple map. There are three colors indicating what needs to be drawn for our floor tiles: Black (0,0,0,255), Blue (0,0,255,255), and Green (0,128,0,255). In our **`TownLevel`** code, we can now just set the tile at that location based on this map.

Code Listing 51: TownLevel.cs

```
// Solid terrain
if (terrainData[w + (h * width)] == Color.Black)
    data.TileType = "Green";
else if (terrainData[w + (h * width)] == Color.Blue)
{
    data.TileType = "Water";
    data.IsSolid = true;
}
else if (terrainData[w + (h * width)] == Color.Green)
    data.TileType = "Grass";
```

As you can see, we have a **`TileData`** object to help manage each tile. We can set its type, which indicates what sprite we want to use in our sprite sheet, and if it is "solid," which indicates whether or not avatars can move through it. Each of our tiles use the same sprite class we created in Chapter 2; this means we can have animated tiles in our level.

This tile terrain would then render the level like this:

Figure 8: Rendered town terrain

If we pass in the other maps for the town, overlay, and object maps (object maps are used for the walls, paths, and buildings), we get the full level rendered.

Figure 9: Rendered town with overlay and object map

Before we get to the code, let's have a look at these other two maps.

Overlay map

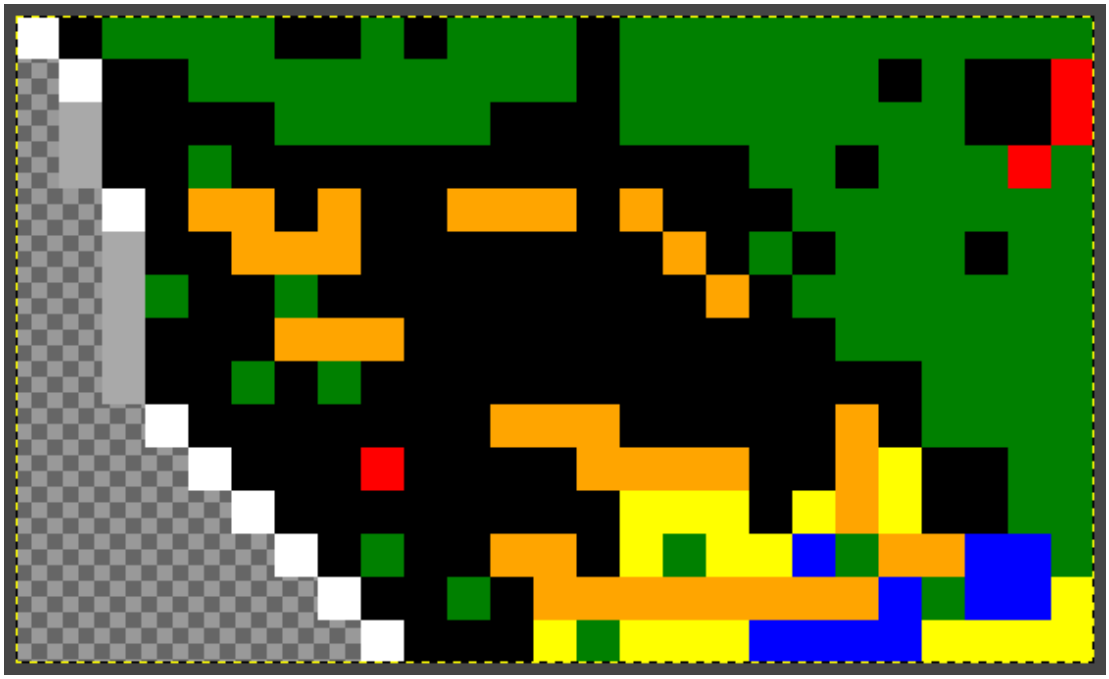


Figure 10: Town overlay map

As you can see, we are using a number of color keys here. Green is now being used for trees, red for mountains, orange for flowers, blue for logs, and yellow for tree stumps. To the left we are using a combination of white and gray to key the cliff sprites to use.

Object map

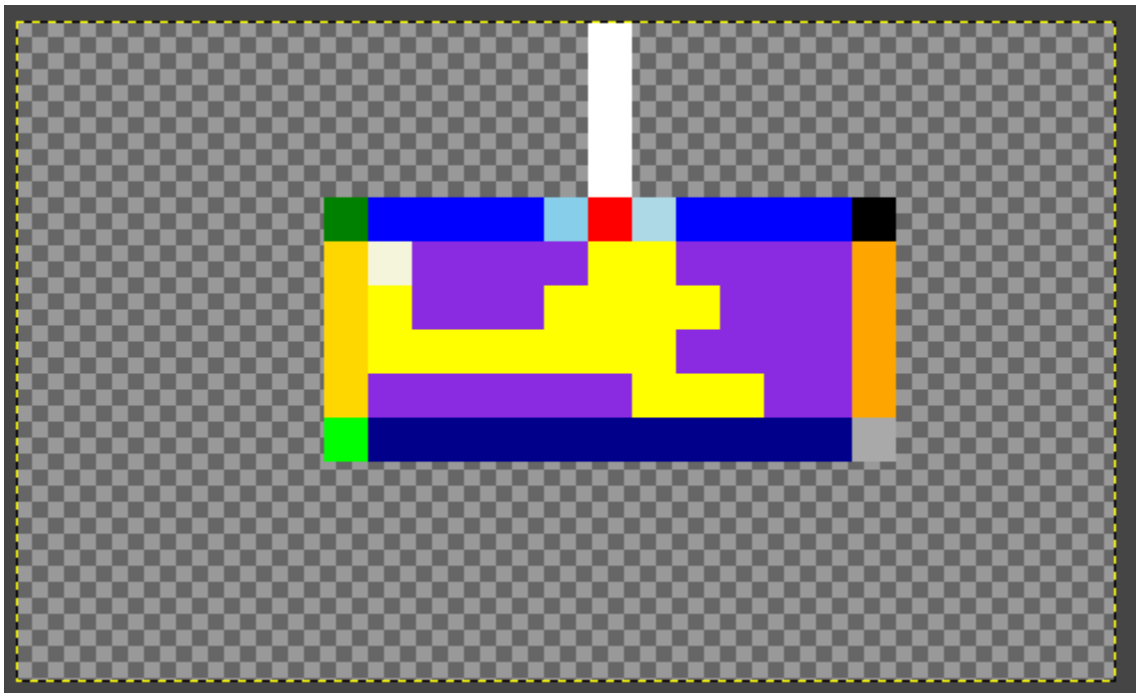


Figure 11: Town object map

Here, we have white for the vertical path, a number of colors to key the walls and their corners, and purple to generate a random house. The ivory in the top left corner of the town is where we want the inn, and the red square is the exit from the town.

LevelBase

All our levels use a common base class. This class will help set up the basic elements of a level and how it is drawn, but our derived classes for each level will have the mapping between the sprite sheet and the tile maps in them.

We have a number of properties in here.

Code Listing 52: LevelBase.cs properties

```
/// <summary>
/// This is a reference to our player in the level
/// </summary>
public virtual Sprite PlayerReference { get; set; }
```

```
/// <summary>
/// This is a list of all our base tiles in the level
/// </summary>
public virtual List<MapTile> Tiles { get; protected set; }
/// <summary>
/// This is a list of all our overlay tiles in the level
/// </summary>
public virtual List<MapTile> OverlayTiles { get; protected set; }

/// <summary>
/// This is the size of each sprite tile in the level
/// </summary>
protected virtual Point tileSize { get; set; }

/// <summary>
/// A reference to the content manager so we can load assets
/// </summary>
protected virtual ContentManager contentManager { get; set; }

/// <summary>
/// The sprite sheet all tiles are rendered from
/// </summary>
protected virtual Texture2D spriteSheet { get; set; }
/// <summary>
/// The tile map used for floor tiles
/// </summary>
protected virtual Texture2D map { get; set; }
/// <summary>
/// The tile map used for overlay objects
/// </summary>
protected virtual Texture2D overlay { get; set; }
/// <summary>
/// The tile map used for overlay objects
/// </summary>
protected virtual Texture2D objects { get; set; }
/// <summary>
/// The tile map used for NPC avatars
/// </summary>
protected virtual Texture2D mobs { get; set; }
```

You can see our tiles are made up of a class called **`MapTile`**. This is used to render the required sprite at a level location and retain the **`TileData`** required for that tile.

Code Listing 53: MapTile class

```
public class MapTile
{
    public float Layer { get; protected set; }
```

```
    public Vector2 Location { get; set; }

    public List<Sprite> Items { get; protected set; }

    public Sprite TileBase { get; set; }

    public TileData Data { get; set; }


    public MapTile(Texture2D spriteSheet, Point tileSize, Point cellSize,
Dictionary<string, SpriteSheetAnimationClip> animation = null, string
initialAnimation = null)
    {

        TileBase = new Sprite(spriteSheet, tileSize, cellSize);

        if (animation != null)
            TileBase.animationPlayer = new
SpriteSheetAnimationPlayer(animation);

        if (initialAnimation != null)
            TileBase.StartAnimation(initialAnimation);
    }

    public virtual void Update(GameTime gameTime)
    {
        if (TileBase != null)
            TileBase.Update(gameTime);
    }

    public virtual void Draw(GameTime gameTime, SpriteBatch spriteBatch)
    {
        if (TileBase != null)
        {
            TileBase.Position = Location;
            TileBase.Draw(gameTime, spriteBatch);
        }
    }
}
```

The class ensures the sprite is updated and drawn in the right location, and the **`Data`** property is an instance of **`TileData`**, which holds the required data for the tile in question.

Code Listing 54: TileData class

```
public class TileData
{
    public bool IsSolid { get; set; }
```

```
    public string TileType { get; set; }
    public string ExitTo { get; set; }
    public Vector2 EnterIn { get; set; }
}
```

This class, as small as it is, is very important. It holds all the required data for a tile in our level: whether it is solid, the type of the tile (this is used to know what sprite is used), if and where the tile exits to another level, when the player enters that level, and at what coordinates.

Let's see how these classes are all used to generate the level.

Code Listing 55: GenerateMap method

```
protected virtual void GenerateMap(Texture2D spriteSheet, Point tileSize,
Point cellSize, Dictionary<Point, TileData> map, Dictionary<Point,
List<TileData>> overlayMap, Dictionary<string, SpriteSheetAnimationClip>
animation)
{
    if (map != null)
    {
        Tiles = new List<MapTile>();
        OverlayTiles = new List<MapTile>();

        foreach (Point p in map.Keys)
        {
            MapTile tile = new MapTile(spriteSheet, tileSize, cellSize,
animation, map[p].TileType);
            tile.Data = map[p];
            tile.Location = new Vector2(p.X, p.Y);

            Tiles.Add(tile);

            if (overlayMap != null && overlayMap.ContainsKey(p))
            {
                foreach (TileData td in overlayMap[p])
                {
                    tile = new MapTile(spriteSheet, tileSize, cellSize,
animation, td.TileType);
                    tile.Data = td;
                    tile.Location = new Vector2(p.X, p.Y);
                    OverlayTiles.Add(tile);
                }
            }
        }
    }
}
```

The derived class will have built and populated the **Dictionary<Point,TileData>** map variable, and our underlying functionality in the abstract class can now put this into our **Tiles** and **OverlayTiles** lists.

Let's now have a look at a derived level class that does the mapping of the sprite sheet to the tile maps and generates this **Dictionary<Point, TileData>** map parameter.

Dungeon level

All we need to do now is create a constructor and the overrides for the abstract methods **GetAnimationClips** and **GenerateTileData** in our derived class.

Deriving from **LevelBase**, our contractor is pretty simple.

Code Listing 56: Dungeon class constructor

```
public Dungeon(ContentManager contentMgr, string tileSheetAsset, string
mapAsset, string overlayMapAsset, string objectMapAsset, string
mobMapAsset, Point tileSize, Point cellSize)
    : base(contentMgr, tileSheetAsset, mapAsset, overlayMapAsset,
objectMapAsset, mobMapAsset, tileSize, cellSize)
{ }
```

We now need to map our sprite sheet, just as we did in Chapter 2.

Code Listing 57: Dungeon animation clips

```
protected override Dictionary<string, SpriteSheetAnimationClip>
GetAnimationClips(Texture2D spriteSheet)
{
    SpriteAnimationClipGenerator sacg = new
SpriteAnimationClipGenerator(new Vector2(spriteSheet.Width,
spriteSheet.Height), new Vector2(4, 9));

    return new Dictionary<string, SpriteSheetAnimationClip>()
    {
        {"Blank", sacg.Generate("Blank", new Vector2(3, 0), new Vector2(3,
0), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"Floor", sacg.Generate("Floor", new Vector2(1, 1), new Vector2(1,
1), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"TLWall", sacg.Generate("TLWall", new Vector2(0, 0), new
Vector2(0, 0), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"LWall", sacg.Generate("LWall", new Vector2(0, 1), new Vector2(0,
1), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"BackWall", sacg.Generate("BackWall", new Vector2(1, 0), new
Vector2(1, 0), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"TRWall", sacg.Generate("TRWall", new Vector2(2, 0), new
Vector2(2, 0), new TimeSpan(0, 0, 0, 0, 500), true) },
```

```
        {"RWall", sacg.Generate("RWall", new Vector2(2, 1), new Vector2(2,
1), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"BLWall", sacg.Generate("BLWall", new Vector2(0, 2), new
Vector2(0, 2), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"BWall", sacg.Generate("BWall", new Vector2(1, 2), new Vector2(1,
2), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"BRWall", sacg.Generate("BRWall", new Vector2(2, 2), new
Vector2(2, 2), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"LBackWall", sacg.Generate("LBackWall", new Vector2(0, 5), new
Vector2(0, 5), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"RBackWall", sacg.Generate("RBackWall", new Vector2(1, 5), new
Vector2(1, 5), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"LFrontWall", sacg.Generate("LFrontWall", new Vector2(1, 6), new
Vector2(1, 6), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"RFrontWall", sacg.Generate("RFrontWall", new Vector2(0, 6), new
Vector2(0, 6), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"BrickOverlay", sacg.Generate("BrickOverlay", new Vector2(2, 5),
new Vector2(2, 5), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"BrickOverlay2", sacg.Generate("BrickOverlay2", new Vector2(3, 5),
new Vector2(3, 5), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"BrickOverlay3", sacg.Generate("BrickOverlay3", new Vector2(2, 6),
new Vector2(2, 6), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"BrickOverlay4", sacg.Generate("BrickOverlay4", new Vector2(3, 6),
new Vector2(3, 6), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"BrickOverlay5", sacg.Generate("BrickOverlay5", new Vector2(3, 7),
new Vector2(3, 7), new TimeSpan(0, 0, 0, 0, 500), true) },
        {"BrickOverlay6", sacg.Generate("BrickOverlay6", new Vector2(3, 8),
new Vector2(3, 8), new TimeSpan(0, 0, 0, 0, 500), true) },

        {"Hole", sacg.Generate("Hole", new Vector2(1, 7), new Vector2(1,
7), new TimeSpan(0, 0, 0, 0, 500), true) },
    };
}
```

Again, we have a **SpriteAnimationClipGenerator** and we map the cells we want from the sprite sheet.

Now, in **GenerateTileData** we can read the tile maps, and based on the given colors in those maps create our map and overlay map dictionaries.

Code Listing 58: Dungeon GenerateTileData

```
protected override void GenerateTileData(Texture2D spriteSheet, Point
tileSize, Point cellSize, Texture2D floorMap, Texture2D overlays, Texture2D
objects, int width, int height, Dictionary<string,
SpriteSheetAnimationClip> animation)
{
    int seed = 1971;
```

```
    Random rnd = new Random(seed);

    Dictionary<Point, TileData> innFloorPlan = new Dictionary<Point,
TileData>();
    Dictionary<Point, List<TileData>> overlay = new Dictionary<Point,
List<TileData>>();

    Color[] floorMapData = new Color[floorMap.Width * floorMap.Height];
    floorMap.GetData(floorMapData);

    Color[] overlayData = new Color[overlays.Width * overlays.Height];
    overlays.GetData(overlayData);

    Color[] objectData = new Color[objects.Width * objects.Height];
    objects.GetData(objectData);


    for (int w = 0; w < width; w++)
    {
        for (int h = 0; h < height; h++)
        {
            TileData data = new TileData();
            Point p = new Point(w * tileSize.X, h * tileSize.Y);

            if (floorMapData[w + (h * width)] == Color.Transparent)
            {
                data.TileType = "Blank";
                data.IsSolid = true;
            }
            else if (floorMapData[w + (h * width)] == Color.White)
            {
                int r = rnd.Next(0, 100);

                if (r <= 25)
                {
                    if (!overlay.ContainsKey(p))
                        overlay.Add(p, new List<TileData>());

                    data = new TileData();

                    r = rnd.Next(0, 100);

                    if (r <= 16)
                        data.TileType = "BrickOverlay";
                    else if (r <= 32)
                        data.TileType = "BrickOverlay2";
                    else if (r <= 48)
                        data.TileType = "BrickOverlay3";
                    else if (r <= 64)
```

```
                    data.TileType = "BrickOverlay4";
                else if (r <= 80)
                    data.TileType = "BrickOverlay5";
                else
                    data.TileType = "BrickOverlay6";

                overlay[p].Add(data);
            }

            data = new TileData();
            data.TileType = "Floor";
        }
        else if (floorMapData[w + (h * width)] == Color.Black)
        {
            data.TileType = "Hole";
            data.ExitTo = "AreaMap";
            data.EnterIn = new Vector2(31, 15);
        }
        else if (floorMapData[w + (h * width)] == Color.Gray)
        {
            data.TileType = "BackWall";
            data.IsSolid = true;
        }
        else if (floorMapData[w + (h * width)] == Color.DarkGray)
        {
            data.TileType = "BWall";
            data.IsSolid = true;
        }
        else if (floorMapData[w + (h * width)] == Color.DimGray)
        {
            data.TileType = "LBackWall";
            data.IsSolid = true;
        }
        else if (floorMapData[w + (h * width)] == Color.DarkSlateGray)
        {
            data.TileType = "RBackWall";
            data.IsSolid = true;
        }
        else if (floorMapData[w + (h * width)] == Color.Red)
        {
            data.TileType = "TLWall";
            data.IsSolid = true;
        }
        else if (floorMapData[w + (h * width)] == Color.RosyBrown)
        {
            data.TileType = "LWall";
            data.IsSolid = true;
        }
        else if (floorMapData[w + (h * width)] == Color.Brown)
```

```
            {
                data.TileType = "BLWall";
                data.IsSolid = true;
            }
            else if (floorMapData[w + (h * width)] == Color.Green)
            {
                data.TileType = "TRWall";
                data.IsSolid = true;
            }
            else if (floorMapData[w + (h * width)] == Color.Lime)
            {
                data.TileType = "RWall";
                data.IsSolid = true;
            }
            else if (floorMapData[w + (h * width)] == Color.LightGreen)
            {
                data.TileType = "BRWall";
                data.IsSolid = true;
            }
            else if (floorMapData[w + (h * width)] == Color.GreenYellow)
            {
                data.TileType = "LFrontWall";
                data.IsSolid = true;
            }
            else if (floorMapData[w + (h * width)] == Color.DarkGreen)
            {
                data.TileType = "RFrontWall";
                data.IsSolid = true;
            }
            else { }
            if (!string.IsNullOrEmpty(data.TileType) &&
!innFloorPlan.ContainsKey(p))
                innFloorPlan.Add(p, data);


        }
    }

    GenerateMap(spriteSheet, tileSize, cellSize, innFloorPlan, overlay,
animation);
}
```

As you can see, each color in the tile maps map to a given sprite for that location in the level. We know all walls are solid, so those have the **`IsSolid`** flags set. The entrance and exit to the **`Level`** in **`Black`** (0,0,0,255) leads back to the **`AreaMap`** at a given location.

Let's have a look at the floor plan tile map for the dungeon.

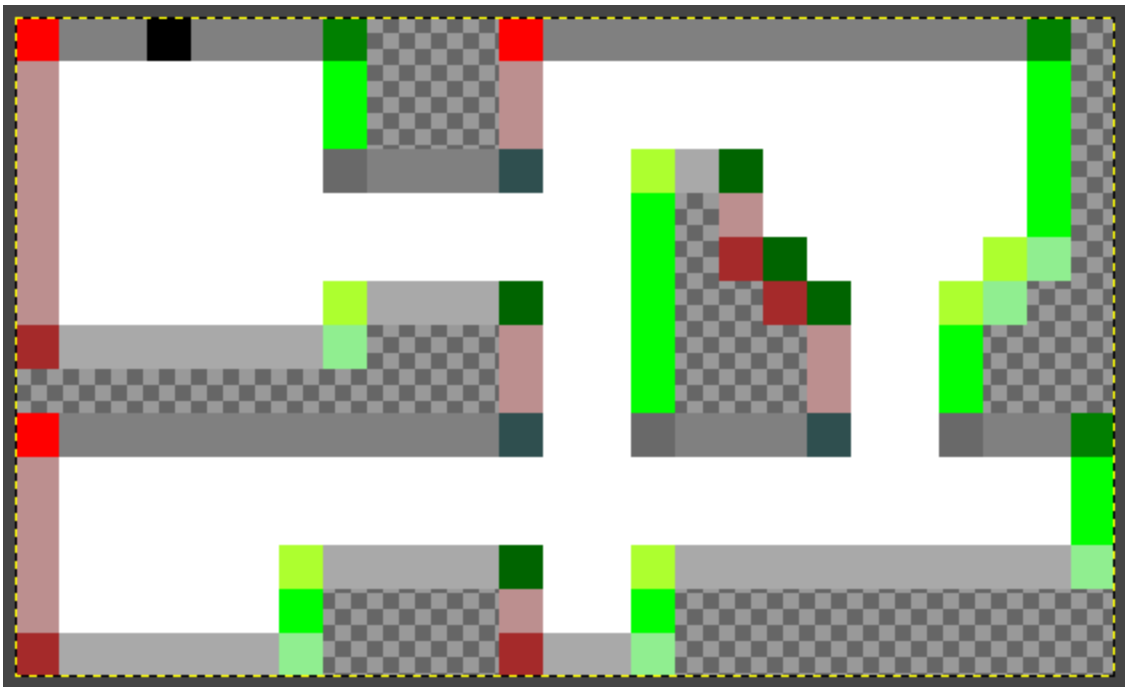


Figure 12: Dungeon terrain map

Looking at the tile map, we can see all red (255,0,0,255) pixels are used to render **`TLWall`** sprites. These are sprites on the sprite sheet that are "top left wall" corners. Looking at the animation map, that is the top left corner of the sprite sheet. The exit in black (0,0,0,255) uses the "Hole" sprite, and is located at (1,7) in the sheet, and we can see that's the open doorway.

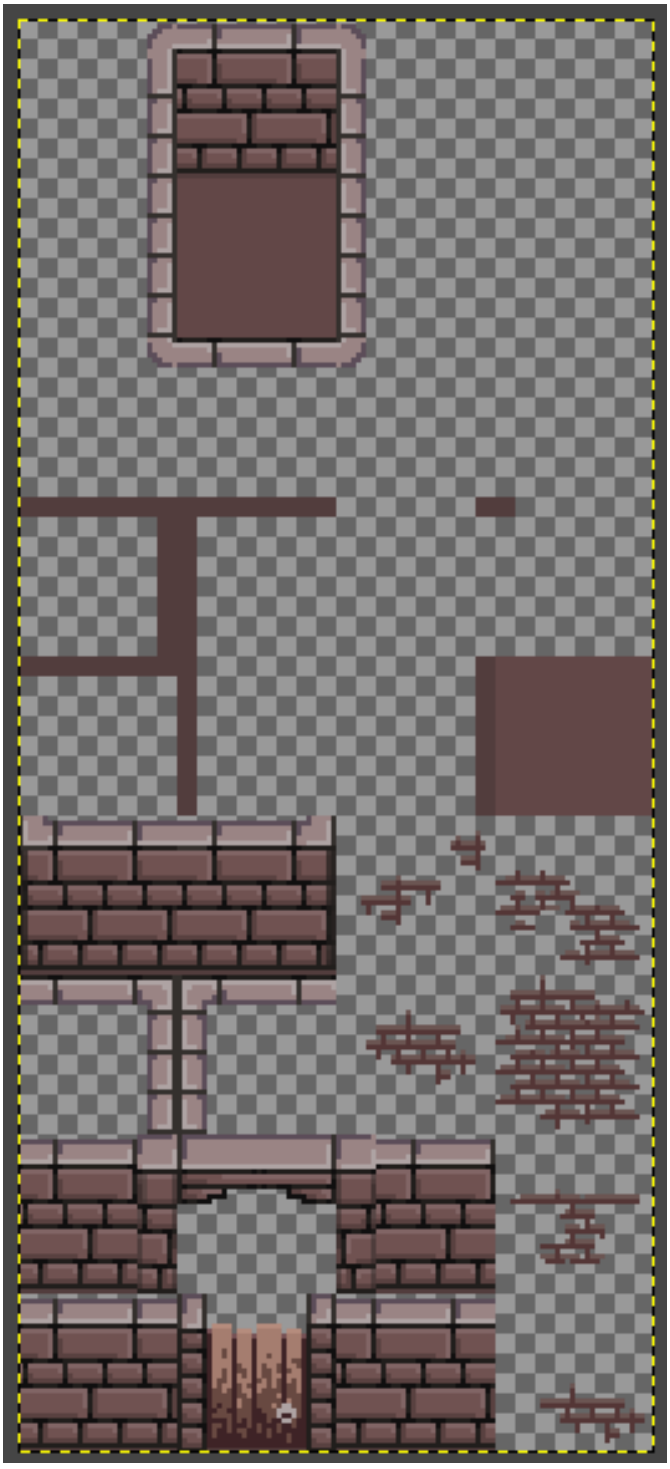


Figure 13: Dungeon sprite sheet

Let’s see how that is rendered.

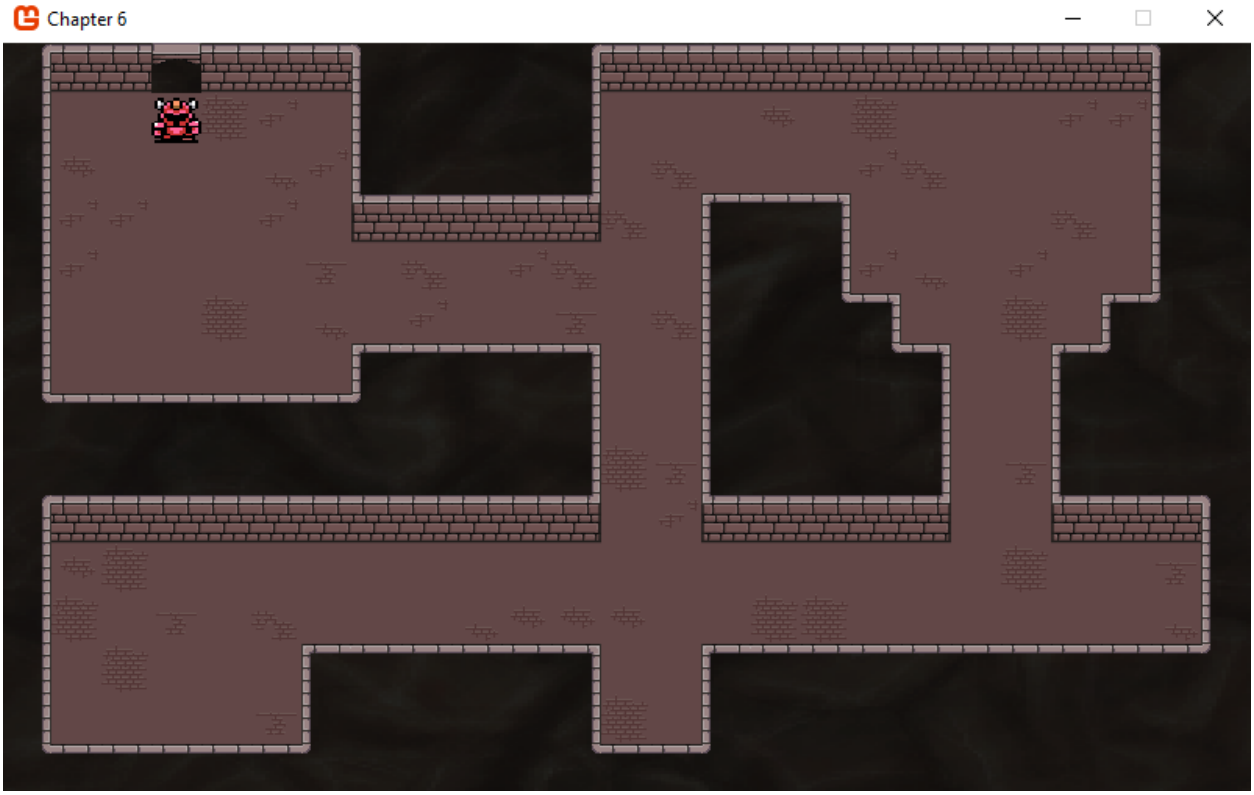


Figure 14: Rendered dungeon terrain

You can now see how to build levels and maps by simply creating a terrain, overlay, and object map for each, and then overriding the **`LevelBase`** class to create the level in-game. Take a look at the **`AreaLevel.cs`** and **`InnLevel.cs`** files; they are built just like **`DungeonLevel.cs`**.

What’s next

We now have a framework for rendering our world, where our character is able to move from one area to the next. In the next chapter, we are going to look at how we can give our character skills and use them in the world.

Chapter 7 Skills

Introduction

We have a character that can walk around the level and interact with some entities, at least to the extent that the character can get quests. If the quest only involves talking to entities, that's fine, but at some point the player is going to have to make their character actually do things—not the least of which is fight something. This is where skills come into play.

Fighting is probably the most used part of a skill system, but it's not everything. Thief characters can probably play most of a game without getting into combat. Sneaking around and stealing stuff without getting caught is what a good thief does, so if the thief has to fight something, that character isn't a very good thief.

Other examples of skills are magic use, which could be used in combat and non-combat situations; crafting to make things like weapons, armor, or anything else that's handmade; and cooking, which could be useful to make potions.

The Skill class

The basis of the skill system starts with the class that encapsulates the data used for skills.

Code Listing 59: Skill class

```
public class Skill
{
    public int ID;
    public String Name;
    public String Description;

    public SkillType Type;

    public Dictionary<int, string> Costs;
    public Dictionary<int,int> ClassBonuses;
    public List<MinMaxBonus> LevelBonuses;
    public Dictionary<int, int> RaceBonuses;
    public Dictionary<int, MinMaxBonus> StatBonuses;

    public bool AlwaysOn;

    public string IconName;
    public Texture2D Icon;
}
```

The **`SkillType`** can be one of the following:

- **Defensive**
- **NonCombat**
- **Offensive**

Defensive and offensive skills are used in combat. Defensive skills include things such as wearing armor or dodging. Non-combat skills are those we've looked at previously.

The **EntitySkill** class is the link between the skill and the entity.

Code Listing 60: EntitySkill class

```
public class EntitySkill
{
    public int ID;
    public int Amount;
}
```

The **Amount** member is the number of ranks the character has with the skill. You can think of ranks as how experienced the character is at using the skill. Code-wise, each rank will increase the chance of success when the character uses that skill. The amount for each rank will vary as more ranks are learned. It should be harder to become a master swordsman, for example, than to learn the basics of sword use, so a character will need more ranks, or the value of a rank will decrease the more the character increases with a skill.

Using skills

Using a skill involves a couple of things: a target, and a difficulty in attempting to use the skill. These two pieces are passed to a method that calculates whether or not the character is successful in using the skill.

Code Listing 61: Method called when using a skill

```
public bool Use(Object target, Entity caster, Difficulty difficulty, out
int result)
{
    int roll;
    int bns = 0;

    roll = GlobalFunctions.GetRandomNumber(DieType.d100);

    roll += (short)difficulty;

    switch (Type)
    {
        case SkillType.Defensive:
        {
            if (target is Entity)
```

```
                roll += ((Entity)target).GetTotalOffBonus();

            break;
        }
        case SkillType.NonCombat:
        {
            if (target is Entity)
                roll += ((Entity)target).GetTotalMiscBonus();

            break;
        }
        case SkillType.Offensive:
        {
            if (target is Entity)
                roll += ((Entity)target).GetTotalDefBonus();

            break;
        }
    }

    //Calculate level bonus
    if (LevelBonuses.Count == 0)
    {
        short level = caster.Level;
        //Use default
        switch (level)
        {
            case 1:
            case 2:
            case 3:
            case 4:
            case 5:
            {
                bns = (short)(10 * level);
                break;
            }
            case 6:
            case 7:
            case 8:
            case 9:
            case 10:
            {
                bns = (short)(50 + (5 * (level - 5)));
                break;
            }
            case 11:
            case 12:
            case 13:
            case 14:
```

```
            case 15:
            {
                bns = (short)(75 + (3 * (level - 10)));
                break;
            }
            case 16:
            case 17:
            case 18:
            case 19:
            case 20:
            {
                bns = (short)(90 + (2 * (level - 15)));
                break;
            }
            default:
            {
                bns = (short)(100 + (1 * (level - 20)));
                break;
            }
        }
    }
    else
    {
        foreach (MinMaxBonus bonus in LevelBonuses)
        {
            if (bonus.IsValueInRange(caster.Level))
            {
                bns += bonus.Amount;
                break;
            }
        }
    }

    roll += bns;

    result = roll;

    if (roll >= 100)
        return true;
    else
        return false;
}
```

The **`Difficulty`** is a simple enum that for our small game has a handful of values.

Code Listing 62: Difficulty enum

```
public enum Difficulty
{
    Impossible = -50,
    VeryHard = -25,
    Hard = -10,
    Normal = 0,
    Easy = 10,
    VeryEasy = 25
}
```

The **`Difficulty`** value is subtracted from a random amount. We also have some optional amounts that can be used to affect the outcome of the attempt.

The target's opposing bonus is subtracted from the total value of the attempt. An offensive attempt is opposed by the target's defensive bonus; a defensive attempt can be opposed by the target's offensive bonus; and a non-combat attempt can be opposed by the target's miscellaneous bonus.

You can give the player a general bonus for the character's level as well. This represents the character being more capable in whatever is attempted as they become more experienced.

If the value of the attempt is 100 or more, the attempt succeeds. The result of the attempt is passed back to whatever code called the method to allow for some flexibility in determining what happens when the attempt succeeds or fails. If the attempt is extraordinarily successful or unsuccessful, you could allow something additional. For example, if the character is fighting and an attack is very successful, you could allow the player to perform something in addition to the attack, perhaps another attack or some extra movement.

Skill example

When the player enters the goblins' cave, they'll discover that the goblins have set a trap to stop invaders. If the character has the ability to detect traps, that skill attempt is resolved. If the result is successful, the character can attempt to disarm the trap if the character has that skill. If the result is not successful, the character will take damage.

The dungeon level will be modified to add a trap near the entrance. The trap will set off an explosion if tripped. Detecting the trap will be pretty easy, however.

There are two steps to adding the trap: modifying the file containing the dungeon layout, and adding code recognizing that modification.

We'll use the color yellow to signify a trap and place it several steps inside the dungeon:

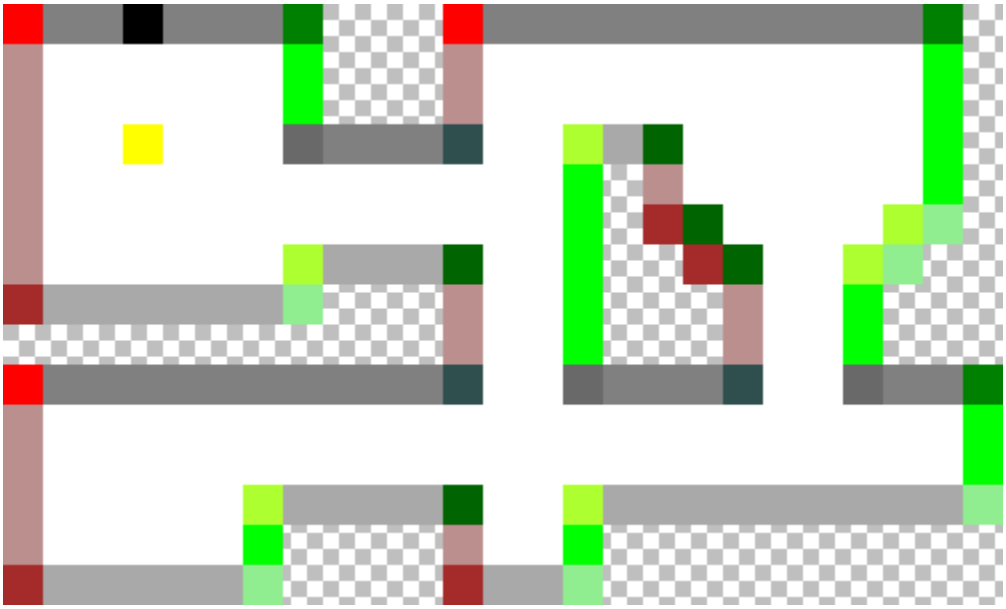


Figure 15: Trapped dungeon level

We need to add a good bit of code to allow the trap to be used: a class for the trap data, a modification to the **Dungeon** class to handle the new pixel, and a check in the update code to recognize when the player steps on the trap to allow the **Detect Trap** skill to be used.

The **Trap** class is very basic at this point. There are many ways to use traps, which you'll know if you're an experienced RPG player, but for our purpose simple is okay.

Code Listing 63: Trap class

```
public class Trap
{
    public TrapType Type { get; set; }
    public int SpellID { get; set; }
    public Difficulty DifficultyLevel { get; set; }
    public int Damage { get; set; }
    public bool Detected { get; set; }
}
```

If the trap is magical, the **SpellID** member is used to tell which spell is cast when the trap is tripped.

Normally, a trap is invisible until detected. Once detected, the **Detected** member is set to allow the code that draws the level to show the trap.

A **Trap** member is added to the **TileData** class to hold the trap information.

The **GameplayScreen** class will have a method added to perform the skill check. It's a sizeable chunk of code, so we won't show it here, but feel free to look at the sample code. Since skills can do many different things, there's no one way to easily handle them.

There's a good bit of code to handle dealing with the result of the **`Detect Trap`** skill if it's unsuccessful and the character is killed. Most of that code can be used for other skill checks as well, notifying the player of the result of the skill attempt.

Implementing other skills

You'll probably want to allow the player to use other skills besides combat-related skills. You'll need to make similar modifications to the **`GameplayScreen`** class as well as the classes for the specific levels. Depending on the type of skill, you may need to modify the level graphic files as well, as we did for the trap.

The first skill you'll probably want to implement is the ability for the character to disarm the trap in the dungeon. You may also want to allow the character to sneak up on the goblin and give the goblin the chance to detect the character.

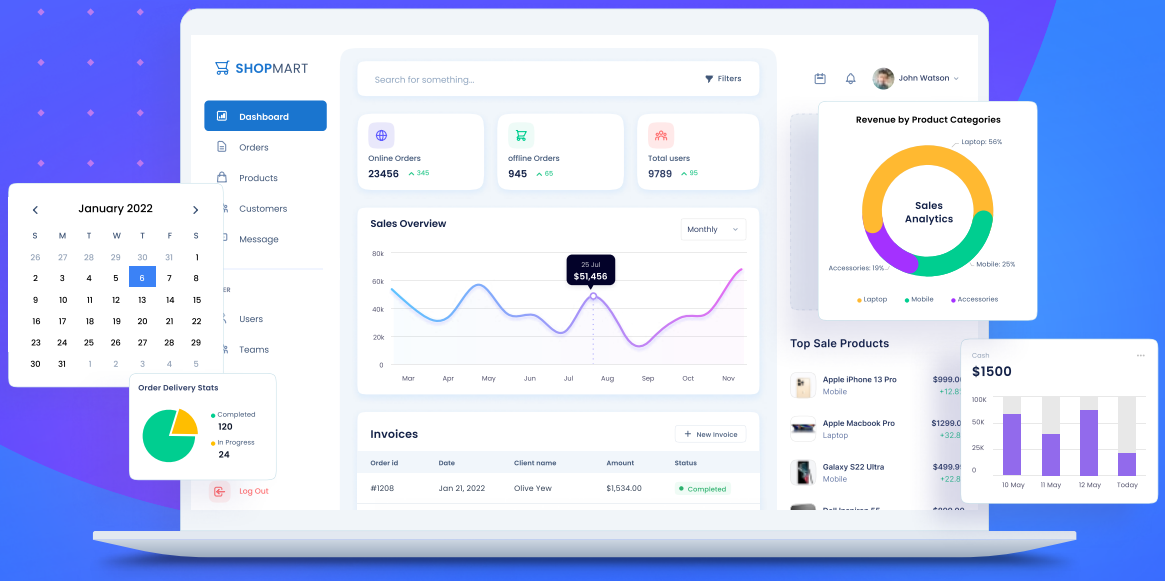
If the character takes damage during the upcoming combat, you may want to enable the character to perform first aid to heal the damage.

What's next

The character can now use their skills to infiltrate the goblin's cave and fight them in order to complete the quest. The next chapter will cover implementing the inventory system that will allow the character to wield weapons and have other items to help in the battle against the goblins.

THE WORLD'S BEST
UI COMPONENT SUITE
FOR BUILDING
POWERFUL APPS

4.6 out of 5 stars



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Apple IBM SIEMENS intel VISA

Syncfusion

Chapter 8 Items and Inventory

In this chapter, we are going to cover items and inventory within our game. The sort of items we will need span from weapons, armor, and equipment, to how we can then store them with backpacks, bags, bottles, and so on.

Item types

In our world, there are going to be a lot of items, and not all of them will be usable by our player. Some may be too big for our character to use, or maybe our characters character class or race prohibits the item's use. Maybe our character can use the item, but with a penalty or even a bonus. We also have to consider if an item can be worn, while it still contributes to the total weight a character is carrying, it is no longer in their inventory as it is equipped.

ItemBase

We have a base item that ALL items will derive from—that's right, we have one base class to rule them all, and in the darkness bind them. (Hmm, I'm sure I've heard that somewhere before…)

Code Listing 64: ItemBase class

```
public class ItemBase : Sprite, IInventoryItem
{
    /// <summary>
    /// Short description of the item.
    /// </summary>
    public string Name { get; set; }
    /// <summary>
    /// Long detailed description of the item.
    /// </summary>
    public string Description { get; set; }
    /// <summary>
    /// The current monetary value of the item.
    /// </summary>
    public decimal Value { get; set; }
    /// <summary>
    /// The weight of the item in kg (easier as it's base 10).
    /// </summary>
    public float Weight { get; set; }

    public float Condition { get; set; }
    /// <summary>
    /// Where, if anywhere, a player can wear/equip the item.
```

```
    /// </summary>
    public EquipableLocation EquipableLocation { get; set; }

    /// <summary>
    /// String markers for the advantages and disadvantages of the item.
    /// </summary>
    public List<string> Mods { get; set; }

    public ItemBase(Texture2D asset, Point size) : base(asset, size, new
Point(16, 16))
    {
        // Set some default values...
        Weight = 0.1f;
        Condition = 1;
    }

}
```

As you can see, we are implementing an interface here too, just to be double sure that all items implement the same properties and methods.

Code Listing 65: IInventoryItem interface

```
public interface IInventoryItem : ISprite
{
    /// <summary>
    /// Short description of the item.
    /// </summary>
    string Name { get; set; }
    /// <summary>
    /// Long detailed description of the item.
    /// </summary>
    string Description { get; set; }
    /// <summary>
    /// The current monetary value of the item.
    /// </summary>
    decimal Value { get; set; }
    /// <summary>
    /// The weight of the item in kg (easier as it's base 10)
    /// </summary>
    float Weight { get; set; }

    float Condition { get; set; }


    List<string> Mods { get; set; }
    /// <summary>
    /// Where, if anywhere, a player can wear/equip the item.
```

```
    /// </summary>
    EquipableLocation EquipableLocation { get; set; }

}
```

All items have a **Name**; this is really the sort name for the item. We can then use **Description** to give a bit more detail about the item. All items also have a **Value**. We have not covered the game's monetary system, so we are keeping it basic with a decimal. You could say that 1 = 1 gold, .5, rather than half a gold, could be 5 silver or 50 copper, and so on.

An item also has a **Weight**. Depending on your game, this could be pounds or kilograms, or just an encumbrance point, or a fraction thereof. An item also has a condition, a percentage value from 0–1. Finally, an item may have some modifications or restrictions applied to it. In the **Mods** list we can store a string representation of what this modification may be. For example, it may be a magic item giving +1 **Strength**, or it may be cursed, giving -1 **Dexterity**.

Weapons

With the previous base class, we can now look at creating other item types, namely weapons, rather than creating a class per weapon (such as a **ShortSword** or **LongBow** class). We can just use a **Weapon** class that can describe any weapon in our game.

Code Listing 66: Weapon class

```
public class Weapon : ItemBase, IWeapon
{
    public string Damage { get; set; }
    public int Range { get; set; }

    public Weapon(Texture2D asset, Point size) : base(asset, size)
    {
        Damage = "D6";
        Mods = new List<string>();
        Range = 1;
    }
}
```

As you can see, we derive from our existing **ItemBase** and implement the **IWeapon** interface.

Code Listing 67: IWeapon interface

```
public interface IWeapon
{
    string Damage { get; set; }
    int Range { get; set; }
}
```

Again, we want to ensure all weapons have the same properties and methods. We have a string to store the **Damage** the weapon can do, so this could be a D6, 2D4, or even D6+1—whatever damage you would like the weapon to do. We can then parse this **Damage** string and inflict the damage accordingly during combat.

We could now create a simple sword like this:

Code Listing 68: Line 64 – GameplayScreen.cs

```
Weapon sword = new
Weapon(_content.Load<Texture2D>("Sprites/Inventory/Sword"), new Point(32,
32))
{
    Value = 55,
    Weight = 5,
    Damage = "D6",
    Name = "Short Sword",
    Description = "This is a basic short sword, nothing special about it.",
    EquipableLocation = EquipableLocation.Hand
};
```

This **Short Sword** has a value of 55 gold, a weight of 5 encumbrance points, and can be equipped in the right or left hand.

We can create a bow like this:

Code Listing 69: Line 94 – GameplayScreen.cs

```
Weapon bow = new Weapon(_content.Load<Texture2D>("Sprites/Inventory/bow"),
new Point(32, 32))
{
    Value = 20,
    Weight = 1,
    Damage = "D4",
    Name = "Bow",
    Description = "A light bow, useless without arrows",
    EquipableLocation = EquipableLocation.TwoHanded,
};
```

Ammunition

Some weapons, like a bow or a sling, will require ammunition; to account for this, we have an **Ammunition** class.

Code Listing 70: Ammunition class

```
public class Ammunition : ItemBase, IAmmunition
{
    public int Quantity { get; set; }
```

```
    public List<string>  Weapons { get; protected set; }

    public Ammunition(Texture2D asset, Point size, params string[] weapons)
: base(asset, size)
    {
        Weapons = new List<string>();
        Weapons.AddRange(weapons);
    }
}
```

Again, this comes with a supporting interface.

Code Listing 71: IAmmunition interface

```
public interface IAmmunition
{
    int Quantity { get; set; }
    List<string> Weapons { get; }
}
```

`Ammunition` has a quantity: how many arrows, bolts, or bullets; and a weapon that this ammunition can be used with. The **`Weapon`** property must match a given **`Weapon Name`** property. The avatar must be in possession of both in order to use the weapon. This only applies to weapons with a range greater than 1.

We can now create some arrows that can be used with our bow.

Armor

We have an offense, but it's nice to have a little defense too, so we also have **`Armor`** items.

Code Listing 72: Armor class

```
public class Armor : ItemBase, IArmor
{
    public int ArmorValue { get; set; }

    public Armor(Texture2D asset, Point size) : base(asset, size)
    {
        ArmorValue = 1;
    }
}
```

Again, a supporting interface comes in tow.

Code Listing 73: IArmor interface

```
public interface IArmor
```

```
{
    int ArmorValue { get; set; }
}
```

We can now create a piece of armor, so let's create a helm.

Code Listing 74: Line 84 – GameplayScreen.cs

```
Armor helm = new Armor(_content.Load<Texture2D>("Sprites/Inventory/helm"),
new Point(32, 32))
{
    Value = 15,
    Weight = 2,
    ArmorValue = 10,
    Name = "Helm",
    Description = "A metal helmet for your head.",
    EquipableLocation = EquipableLocation.Head,
};
```

Next, let's create some boots.

Code Listing 75: Line 74 – GameplayScreen.cs

```
Armor boots = new
Armor(_content.Load<Texture2D>("Sprites/Inventory/boots"), new Point(32,
32))
{
    Value = 5,
    Weight = 2,
    ArmorValue = 3,
    Name = "Boots",
    Description = "Some basic leather boots, they will protect your feet.",
    EquipableLocation = EquipableLocation.Feet,
};
```

Consumables

Again, we can now derive from **`ItemBase`** and, with the enforcing interface, create a class for consumables.

Code Listing 76: ConsumableItem class

```
public class ConsumableItem : ItemBase, IConsumable
{
    public int Quantity { get; set; }
```

```
    public ConsumableItem(Texture2D asset, Point size) : base(asset, size)
{ }
}
```

As mentioned, here's the interface to go with it:

Code Listing 77: IConsumable interface

```
public interface IConsumable
{
    public int Quantity { get; set; }
}
```

Just as with the other items' types, let's now create a potion. It will have three sips: the first two will heal for **D6 HP**, and the last for **D4 HP**.

Code Listing 78: Line 112 – GameplayScreen.cs

```
ConsumableItem healingPotion = new
ConsumableItem(_content.Load<Texture2D>("Sprites/Inventory/healingPotion"),
new Point(32, 32))
{
    Value = 100,
    Weight = 1,
    Quantity = 3,
    Mods = new List<string> { "D6 HP", "D6 HP", "D4 HP" },
    Name = "Healing Potion",
    Description = "3 sips heals D6, D6, and finally D4 HP."
};
```

Keys

By now you should be able to see a pattern. We can create pretty much any kind of item we want with our game now, so let's look at creating a key. We may need things like keys to open doors and chests in our game—who knows where our scenario will take our players.

Code Listing 79: Key class

```
public class Key : ItemBase, IKey
{
    public long LockID { get; set; }

    public Key(Texture2D asset, Point size) : base(asset, size) { }
}
```

We have a class for **Keys**, and it has a supporting interface.

Code Listing 80: IKey interface

```
public interface IKey
{
    public long LockID { get; set; }
}
```

The **LockID** means that the key will only fit a given lock; your chests or doors will need to have the corresponding ID for the key to work on them.

Let's create a key.

Code Listing 81: Line 132 – GameplayScreen.cs

```
Key key1 = new Key(_content.Load<Texture2D>("Sprites/Inventory/key"), new
Point(32, 32))
{
    Name = "A rusty old key.",
    Description = "This key will open the door to Grendel's house...",
    LockID = 12345
};
```

Inventory

We now need a way to store and render the items in our game. For this we have created an **Inventory** base class, **InventoryBase**.

This is quite a big class, so I am not going to post it all here; let's just have a look at how it is defined and at some of its methods.

Code Listing 82: InventoryBase class

```
public class InventoryBase : IInventoryContainer
{

    public string Name { get; set; }

    public string Description { get; set; }
    public Sprite InvSlotBox { get; set; }

    public Vector2 Position { get; set; }

    public Sprite InventoryContainer { get; set; }

    public int invSize = 8;
    public int invBox = 32;
    public float invVPos = 0;
```

```
    int maxLines = 0;

    protected IInventoryItem mouseOver = null;
    public IInventoryItem SelectedItem = null;

    Sprite up, down;

    Texture2D scrollBarRect;

    public SpriteFont Font { get; set; }
    public SpriteFont tinyFont { get; set; }

    public int? MaxVolume { get; set; }

    public List<IInventoryItem> Items { get; set; }

    public float TotalWeight { get; set; }

    protected Sprite closeButton { get; set; }

    public bool IsShowing { get; set; }
```

As we can see, there are a number of properties in here, from the name and description of the inventory this will manage; to screen position and sprites used for its container, inventory box dimensions, and sizes; current selected items; sprites for navigation and scrolling the inventory; a list of all the items in it and total weight of all the items; and some other render elements.

Let's have a look at the constructor for it.

Code Listing 83: InventoryBase constructor

```
public InventoryBase(Texture2D background, Texture2D slotBox, SpriteFont
font, SpriteFont fontSmall, Texture2D upButton, Texture2D downButton,
Texture2D closeBtn = null)
{
    Font = font;
    tinyFont = fontSmall;
    Items = new List<IInventoryItem>();

    InventoryContainer = new Sprite(background, new Point(32 * invSize +
32, 255), new Point(512, 512));
    InventoryContainer.Tint = Color.Cyan;

    InvSlotBox = new Sprite(slotBox, new Point(invBox, invBox), new
Point(64, 64));
    InvSlotBox.Tint = Color.DarkCyan;
```

```
    up = new Sprite(upButton, new Point(16, 16), new Point(64, 64));
    down = new Sprite(downButton, new Point(16, 16), new Point(64, 64));

    if (closeBtn != null)
    {
        closeButton = new Sprite(closeBtn, new Point(16, 16), new
Point(closeBtn.Width, closeBtn.Height));
        closeButton.Tint = Color.White;
    }

    IsShowing = true;
}
```

As we can see, we are passing in the texture to be used for the background, item slot boxes, and the fonts and textures for navigation buttons.

As ever, we have an interface to support this base class, **`IInventoryContainer`**.

Code Listing 84: IInventoryContainer interface

```
public interface IInventoryContainer
{
    /// <summary>
    /// Maximum number of slots. NOTE this is a nullable int, a null means
that there is no limit
    /// </summary>
    int? MaxVolume { get; set; }

    public string Name { get; set; }
    public string Description { get; set; }

    void AddItem(IInventoryItem item);
    void RemoveItem(IInventoryItem item);
    void HandleInput(GameTime gameTime, PlayerIndex? playerIndex,
InputState input);
    void Draw(GameTime gameTime, SpriteBatch spriteBatch);
}
```

With this we can create an inventory container for our items. The **`Inventory`** base class will also handle rendering our items when we wish to view them. The idea is that this base class can be used to show the contents of any inventory: the player's inventory, items in a shop, or even items in a bag or chest the player is examining. We can add and remove items. The base class even has some basics for handling interaction with the mouse. We can have the inventory limited to a max number of slots using the **`MaxVolume`** property, or leave it **`null`** for an infinite number of items.

As you can see, the inventory system also keeps track of the total encumbrance points in it.

AddItem

Code Listing 85: InventoryBase AddItem method

```
public void AddItem(IInventoryItem item)
{
    if (MaxVolume == null || Items.Count + 1 < MaxVolume.Value)
        Items.Add(item);
}
```

Quite simply, if we have no max limit, or our current count +1 is lower than the max limit, we add the item to the inventory **`Items`** list.

RemoveItem

Code Listing 86: InventoryBase RemoveItem method

```
public void RemoveItem(IInventoryItem item)
{
    Items.Remove(item);
}
```

If the item is in the list, it's removed.

Player inventory

We now have items and a basic inventory system, so let's implement an inventory system for our hero.

This is another busy class, so let's break it down a bit at a time. We have a number of properties for this class to support the rendering of the player inventory, its position, the sprites to be used for rendering equipped items, as well as the background for the render, fonts to be used, and the slot box sprite used to render inventory items in.

We also have properties to help with calculating the current encumbrance, as well as other rendering elements. The two key properties in here though are instances of **`InventoryBase`**; this is where we will keep our player's equipment and the **`Equipped`** dictionary. This dictionary holds the items the character is actually using or wearing.

To denote the item slots on a character, we set the dictionary up in the constructor for the class.

Code Listing 87: Line 64 – PlayerInventory.cs

```
// Equipable slots
Equipped = new Dictionary<EquipableLocation, IInventoryItem>()
{
    {  EquipableLocation.Head, null },
    {  EquipableLocation.Neck, null },
    {  EquipableLocation.Body, null },
```

```
    {  EquipableLocation.Chest, null },
    {  EquipableLocation.Abdomen, null },
    {  EquipableLocation.Left_Arm, null },
    {  EquipableLocation.Right_Arm, null },
    {  EquipableLocation.Left_Leg, null },
    {  EquipableLocation.Right_Leg, null },
    {  EquipableLocation.Feet, null },
    {  EquipableLocation.TwoHanded, null },
    {  EquipableLocation.Left_Hand, null },
    {  EquipableLocation.Right_Hand,null }
};
```

We now have a number of functions we can use to place items in the inventory, to then use or equip those items as well as unequip an equipped item. And finally, we can drop items we no longer wish to carry.

PickUp

Code Listing 88: Line 268 – PlayerInventory.cs

```
public string Pickup(IInventoryItem item)
{
    if (item != null)
    {
        item.Size = Inventory.InvSlotBox.Size;
        Inventory.AddItem(item);

        return $"You have picked up {item.Name}";
    }
    else
        return "Nothing to pick up...";
}
```

The method is pretty simple in itself: it first checks if the item we are trying to pick up is not null, then sizes the item so that it renders correctly in the UI for the inventory system and adds the item to the inventory system.

Drop

Code Listing 89: Line 281 – PlayerInventory.cs

```
public string Drop(IInventoryItem item)
{
    Inventory.RemoveItem(item);
    DroppedItem = item;

    return $"You have dropped {item.Name}";
}
```

Again, it's simple enough: remove the item from the inventory system, then set the **`DroppedItem`** to the item dropped. This can then be used by the level we are in to transfer the item to it.

EquipItem

Code Listing 90: Line 304 – PlayerInventory.cs

```
public string EquipItem(IInventoryItem item)
{
    EquipableLocation location = Inventory.SelectedItem.EquipableLocation;

    if (location != EquipableLocation.None)
    {

        if (location == EquipableLocation.Hand)
        {
            // Is there anything in either hand?
            if (Equipped[EquipableLocation.Left_Hand] == null ||
Equipped[EquipableLocation.Right_Hand] == null)
            {
                if (Equipped[EquipableLocation.Left_Hand] == null)
                    location = EquipableLocation.Left_Hand;
                else
                    location = EquipableLocation.Right_Hand;
            }
            else
                location = EquipableLocation.Right_Hand;
        }

        if (Equipped.ContainsKey(location))
        {
            UnEquip(location);

            if (location == EquipableLocation.TwoHanded)
            {
                UnEquip(EquipableLocation.Left_Hand);
                UnEquip(EquipableLocation.Right_Hand);
            }
            else if (location == EquipableLocation.Left_Hand || location ==
EquipableLocation.Right_Hand)
            {
                if (Equipped[EquipableLocation.TwoHanded] != null)
                    UnEquip(EquipableLocation.TwoHanded);
            }

            Equipped[location] = Inventory.SelectedItem;
            Inventory.RemoveItem(Inventory.SelectedItem);

            Inventory.SelectedItem = null;
```

```
            return $"{Equipped[location].Name} has been equipped...";
        }
        else
            return "You can't equip this item.";
    }
    else
    {
        if (Inventory.SelectedItem is IInventoryContainer)
        {
            // Render the container...
            currentContainer = (IInventoryContainer)Inventory.SelectedItem;
        }
    }

    return "You can't equip this item.";
}
```

We first find out the location that this item can be equipped at. If the location is **Hand**, then we need to find out which hand, if any, is free to be allocated the item. Once that's done, we can then equip the item.

First, we unequip any item that might already be equipped there, then we check if the item is two-handed, like a bow or a two-handed sword. If it is two-handed, then we clear out both the left and right hands, as we need to hold such items with both hands.

If it's not two-handed but can be equipped for either hand, we check if the **TwoHanded** slot has something equipped. If it does, then we unequip it as we are now equipping one of the hands with a new item, and so the two-handed item can now no longer be equipped.

Once all location combinations are sorted out, we set the **Equipped** slot to this item. The item is now equipped and in use, so we remove the item from the inventory system and clear the selected inventory item.

UnEquip

Code Listing 91: Line 289 – PlayerInventory.cs

```
public string UnEquip(EquipableLocation targetLocation)
{
    if (Equipped[targetLocation] == null)
        return $"There is nothing to remove here.";

    IInventoryItem item = Equipped[targetLocation];
    Equipped[targetLocation] = null;

    // Add it back to inventory.
    Pickup(item);
```

```
    return $"You are no longer using {item.Name}";
}
```

Again, this is a nice, simple method. First, we check if there is something at the location we are unequipping, then we get the item, set the equipped slot to null to unequip it, and call the **`Pickup`** method with this item to move it back into the inventory system.

Example

We now have an item and inventory framework and a player inventory system. We can create a few items and have the player inventory system pick them up.

Code Listing 92: Line 154 – GameplayScreen.cs

```
playerInventory.Pickup(sword);
playerInventory.Pickup(boots);
playerInventory.Pickup(helm);
playerInventory.Pickup(bow);
playerInventory.Pickup(arrows);
playerInventory.Pickup(healingPotion);
playerInventory.Pickup(shield);
playerInventory.Pickup(key1);
```

Having created the items previously, we can now add them to the player's inventory with the **`Pickup`** method. When we render the inventory system, it looks like this:

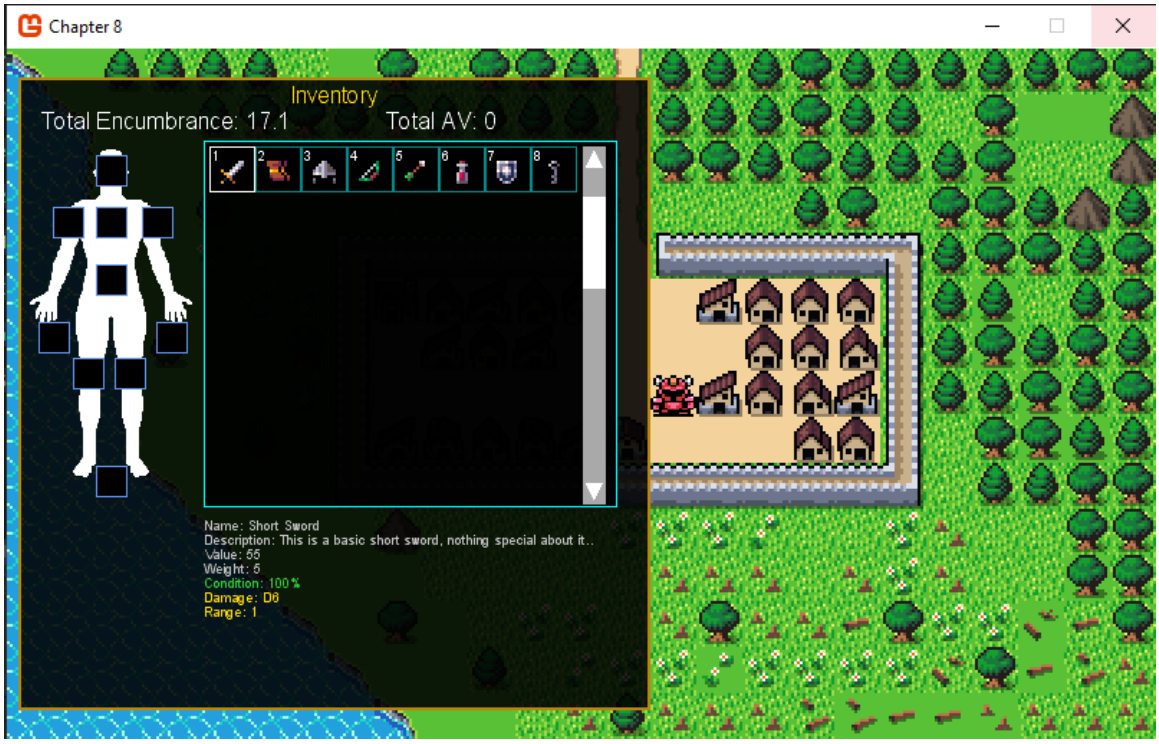


Figure 16: Rendered player inventory

I can use the arrow keys to navigate around the inventory items and use the mouse to click an item to select it. Moving to or selecting the shield looks like this:

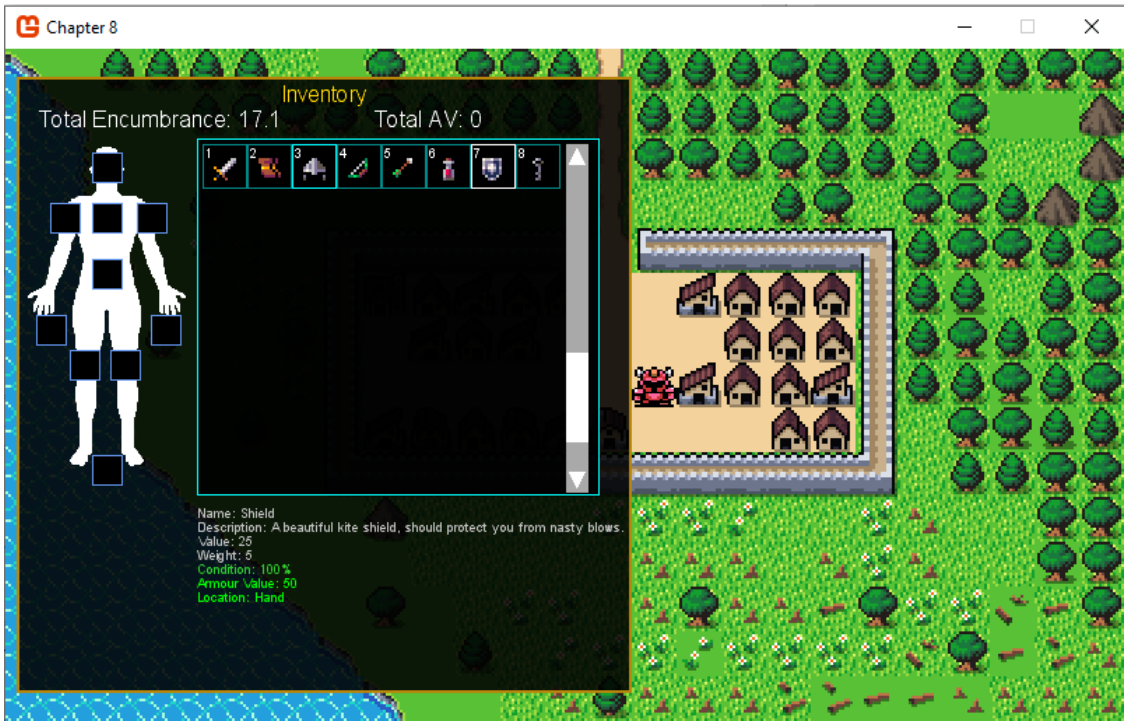


Figure 17: Navigating player inventory

With that item selected, I can press **Enter** to equip it.

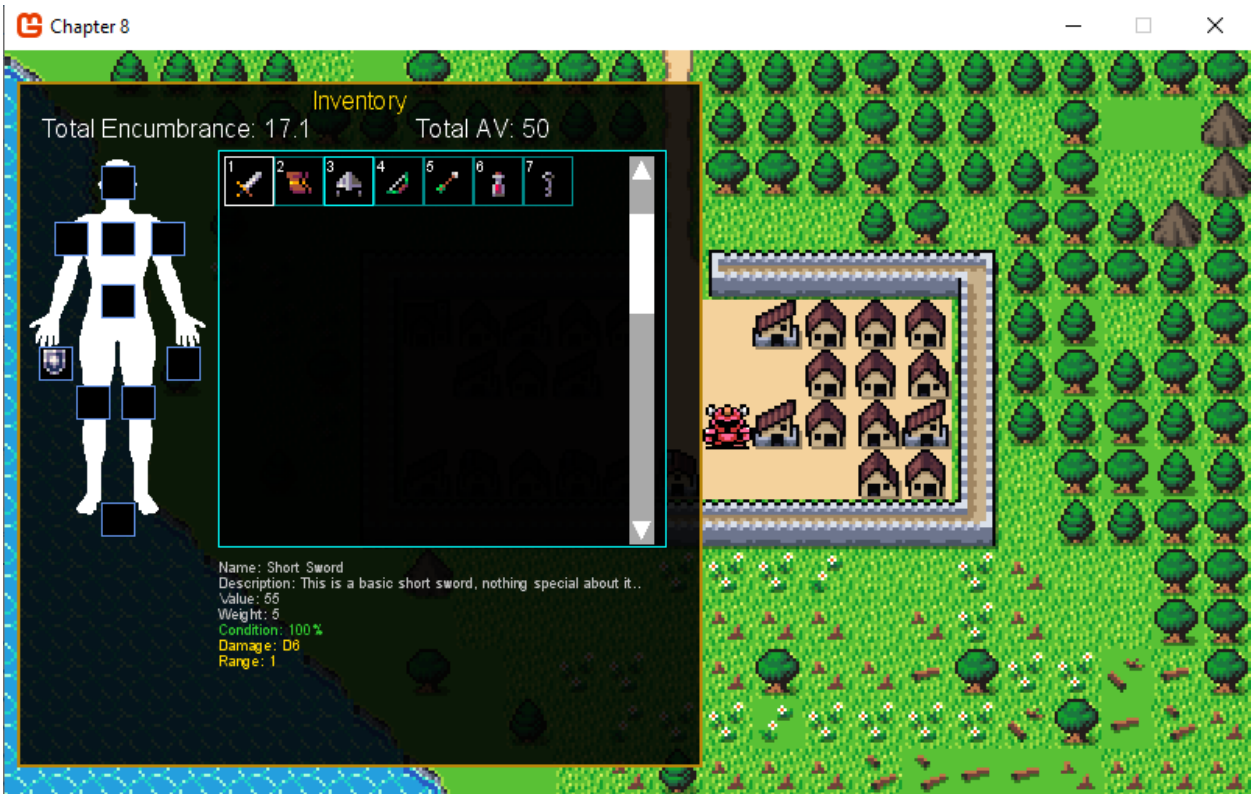


Figure 18: Rendered equipped item

If I hover my mouse over the equipped item, I can see where and what it is.

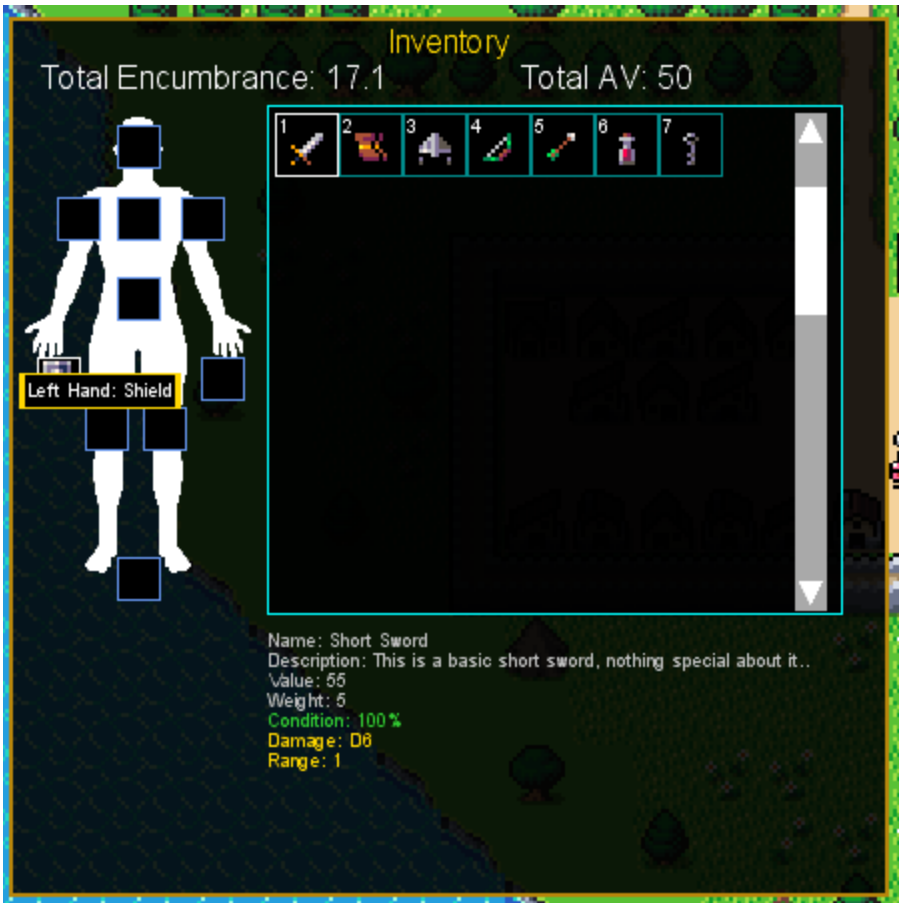


Figure 19: Player inventory hover

I can also press the **D** key to drop the selected item. Let's drop the key.

Figure 20: Dropped the key

As you can see, I have moved my avatar off the spot where I dropped the key, but how does the level know we have dropped an item here and how do we render it?

Within the **`LevelBase`** class, I have added a **`Name`** property for the levels; each level now also sets its name in the constructor. I have also added a static dictionary to keep track of what items have been left in what levels.

Code Listing 93: Line 63 – LevelBase.cs

```
    static Dictionary<string, List<IInventoryItem>> Items { get; set; }

    protected string Name { get; set; }
```

An **`AddItem`** method allows us to add items to the level at runtime; this is done when an item is dropped.

Code Listing 94: AddItem method added to LevelBase class

```
public void AddItem(IInventoryItem item)
{
    if (Items == null)
        Items = new Dictionary<string, List<IInventoryItem>>();
```

```
    if (!Items.ContainsKey(Name))
        Items.Add(Name, new List<IInventoryItem>());

    item.Position = new
Vector2((int)PlayerReference.Position.X,(int)PlayerReference.Position.Y);
    item.Size = PlayerReference.Size;
    Items[Name].Add(item);
}
```

A **PickUpItem** method also allows for items in the level to be picked up by the player.

Code Listing 95: PickupItem method added to LevelBase class

```
public IInventoryItem PickupItem()
{
    if (Items != null)
    {
        IInventoryItem item = Items[Name].FirstOrDefault(i => new
Rectangle((int)i.Position.X, (int)i.Position.Y, i.Size.X,
i.Size.Y).Intersects(PlayerReference.BoundsRectangle));

        if (item != null)
        {
            Items[Name].Remove(item);
            return item;
        }
    }

    return null;
}
```

And finally, in the **LevelBase** class, we can render any items we have in the level by adding the following code to the **LevelBase Draw** method.

Code Listing 96: Added code to draw item in LevelBase class

```
public virtual void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    if (Tiles != null)
    {
        foreach (MapTile tile in Tiles)
            tile.Draw(gameTime, spriteBatch);
    }

    if (OverlayTiles != null)
    {
        foreach (MapTile tile in OverlayTiles)
            tile.Draw(gameTime, spriteBatch);
```

```
    }

    if (Items != null && Items.ContainsKey(Name))
    {
        foreach (IInventoryItem item in Items[Name])
            ((ItemBase)item).Draw(gameTime, spriteBatch);
    }

    PlayerReference.Draw(gameTime, spriteBatch);
}
```

The render

InventoryBase render

Let's have a look at how the inventory is rendered. Since we can render an infinite number of items, we don't want the items to fill the screen or overflow from the area we are rendering in. That's where the **`InventoryContainer`** comes in; this sprite is going to represent where we are going to be rendering our items.

We are using the **`ScissorRectangle`** in the sprite batch to cull pixels that are rendered outside of the **`InventoryContainer`**. In order to do this, we have to "interrupt" the sprite batch **`Draw`** call.

We are using the current screen's sprite batch to render, and this has already started a **`Begin`** call. To use the **`ScissorRectangle`**, we need to end the current call, begin with the **`ScissorRectangle`** data we need, draw our items, end again, and then begin again so the rest of the current screen's drawable items can continue to be rendered.

First, we end the current **`Draw`** call and start a new one with our **`ScissorRectangle`** data.

Code Listing 97: Setting up the ScissorRectangle

```
spriteBatch.End();
spriteBatch.Begin(SpriteSortMode.Immediate, BlendState.AlphaBlend,
SamplerState.PointClamp, DepthStencilState.DepthRead, new RasterizerState()
{ ScissorTestEnable = true, });
spriteBatch.GraphicsDevice.ScissorRectangle =
InventoryContainer.BoundsRectangle;
```

Now, anything rendered outside of the **`ScissorRectangle`** will be culled and not rendered to the screen. We can now loop through all the items we have in the **`Items`** list and draw them in place, not caring if they overflow as they will be culled.

Code Listing 98: Render the Items

```
foreach (IInventoryItem item in Items)
{
    // c is the current item count, starting at 0.
    int l = c / invSize;
    int nl = l % invSize;
    Vector2 itemPos = Position + new Vector2(4 + ((c * invBox) - (l *
invSize * invBox)), 4 + (l * invBox));

    if (l == 0)
        maxLines++;

    itemPos.Y += invVPos;

    if (item == SelectedItem)
        InvSlotBox.Tint = Color.White;
    else if (item == mouseOver)
        InvSlotBox.Tint = Color.Cyan;
    else
        InvSlotBox.Tint = Color.DarkCyan;

    InvSlotBox.Position = itemPos;
    InvSlotBox.Draw(gameTime, spriteBatch);

    // Render Item in it...
    if (item is ItemBase)
    {
        item.Position = itemPos;
        ((ItemBase)item).Draw(gameTime, spriteBatch);
    }

    c++;

    spriteBatch.DrawString(tinyFont, $"{c}", itemPos + new Vector2(2, 2),
Color.White);
}
```

We can then end our **`Draw`** call and begin again so the screen can continue to render its items, and we can render outside of the **`ScissorRectangle`**.

Code Listing 99: End ScissorRectangle render

```
spriteBatch.End();
spriteBatch.Begin(SpriteSortMode.Immediate, BlendState.AlphaBlend,
SamplerState.PointClamp, DepthStencilState.DepthRead);
```

Let's see how that would render 1,000 items by making 1,000 swords and putting them in the system like this:

Code Listing 100: 1,000 swords in the inventory

```
for (int i = 0; i < 1000; i++)
{
    Weapon tst = new
Weapon(_content.Load<Texture2D>("Sprites/Inventory/Sword"), new Point(32,
32))
    {
        Value = 55,
        Weight = 5,
        Damage = "D6",
        Name = $"Short Sword [{i}]",
        Description = "This is a basic short sword, nothing special about
it..",
        EquipableLocation = EquipableLocation.Hand
    };

    playerInventory.Pickup(tst);
}
```

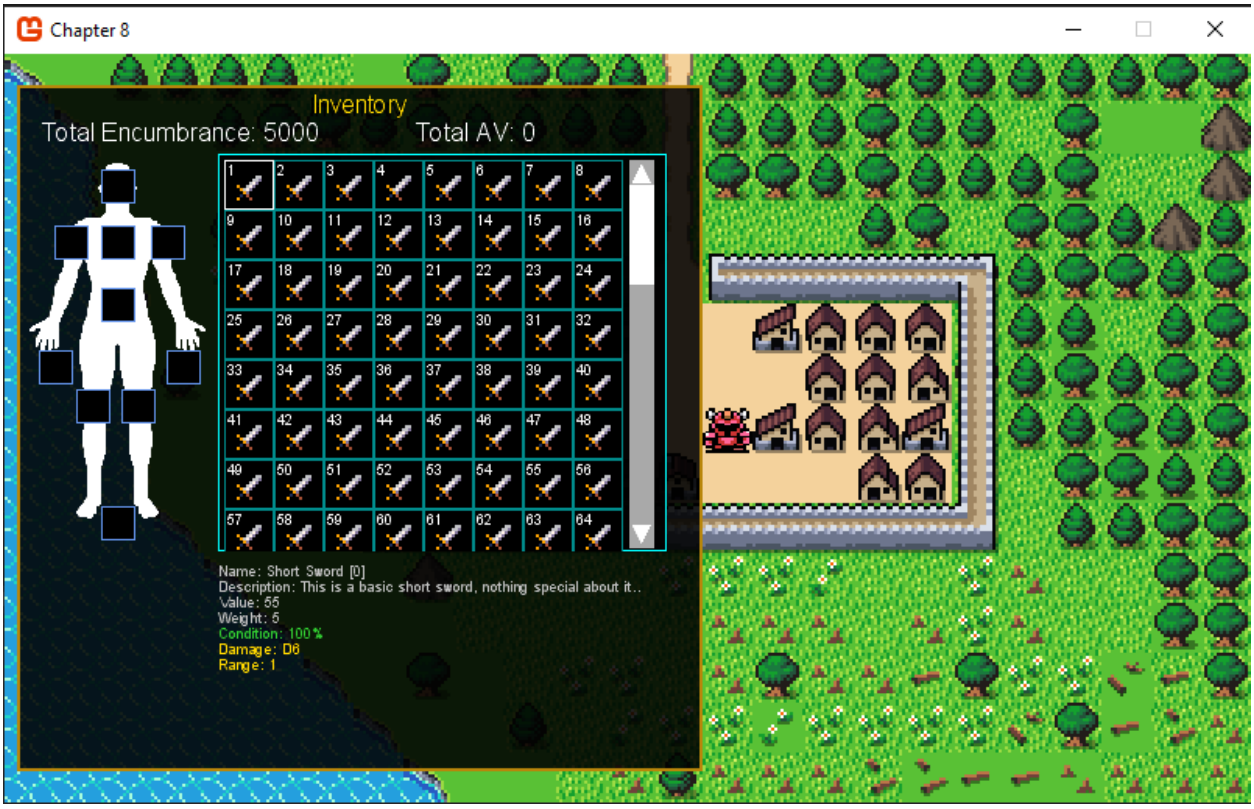


Figure 21: 1,000 Swords culled by the ScissorRectangle

As you can see, all the items are rendered and any that are rendered outside the **`ScissorRectangle`** are culled. Let's scroll down with the arrow keys a bit to ensure that items that are moved up the screen are also culled.

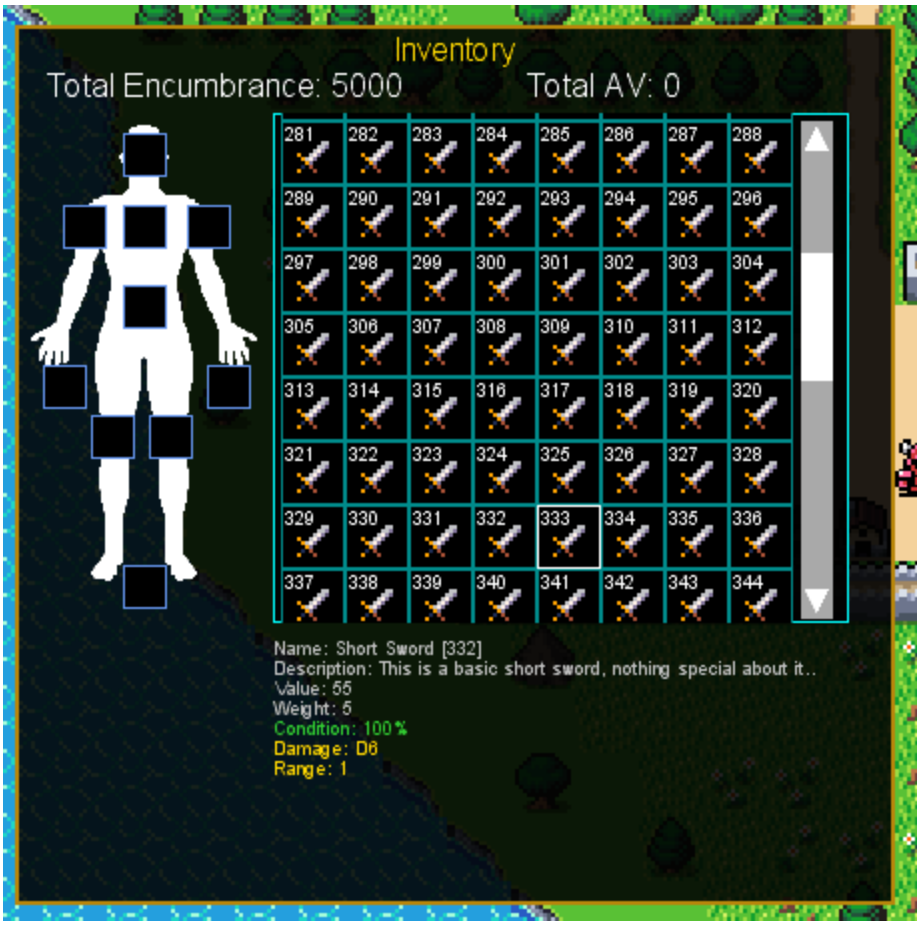


Figure 22: Scrolling 1,000 swords culled

Yep, that works great!

But what if we didn't set the scissor rectangle? What would it look like? Let's give it a go.

Figure 23: Scrolling 1,000 swords, no culling

As expected, it's a mess. Good thing we have that **`ScissorRectangle`**.

You can also see that we are rendering information on the current selected item below the container. This is done based on the type of item being viewed.

Code Listing 101: Rendering selected item information

```
if (SelectedItem != null)
{
    spriteBatch.DrawString(tinyFont, $"Name: {SelectedItem.Name}", Position
+ new Vector2(0, baseY), baseColor);
    spriteBatch.DrawString(tinyFont, $"Description:
{SelectedItem.Description}", Position + new Vector2(0, baseY +
tinyFont.LineSpacing), baseColor);
    spriteBatch.DrawString(tinyFont, $"Value: {SelectedItem.Value}",
Position + new Vector2(0, baseY + tinyFont.LineSpacing * 2), baseColor);
    spriteBatch.DrawString(tinyFont, $"Weight: {SelectedItem.Weight}",
Position + new Vector2(0, baseY + tinyFont.LineSpacing * 3), baseColor);

    Color conditionColor = Color.Lerp(Color.Red, Color.LimeGreen,
SelectedItem.Condition);
```

```
    spriteBatch.DrawString(tinyFont, $"Condition:
{(int)(SelectedItem.Condition * 100f)}%", Position + new Vector2(0, baseY +
tinyFont.LineSpacing * 4), conditionColor);
}
else
{
    spriteBatch.DrawString(tinyFont, $"Name: ", Position + new Vector2(0,
baseY), baseColor);
    spriteBatch.DrawString(tinyFont, $"Description: ", Position + new
Vector2(0, baseY + tinyFont.LineSpacing), baseColor);
    spriteBatch.DrawString(tinyFont, $"Value: ", Position + new Vector2(0,
baseY + tinyFont.LineSpacing * 2), baseColor);
    spriteBatch.DrawString(tinyFont, $"Weight: ", Position + new Vector2(0,
baseY + tinyFont.LineSpacing * 3), baseColor);
    spriteBatch.DrawString(tinyFont, $"Condition: %", Position + new
Vector2(0, baseY + tinyFont.LineSpacing * 4), baseColor);
}

if (SelectedItem is IWeapon)
{
    spriteBatch.DrawString(tinyFont, $"Damage:
{((IWeapon)SelectedItem).Damage}", Position + new Vector2(0, baseY +
tinyFont.LineSpacing * 5), Color.Gold);
    spriteBatch.DrawString(tinyFont, $"Range:
{((IWeapon)SelectedItem).Range}", Position + new Vector2(0, baseY +
tinyFont.LineSpacing * 6), Color.Gold);
}

if (SelectedItem is IArmor)
{
    spriteBatch.DrawString(tinyFont, $"Armour Value:
{((IArmor)SelectedItem).ArmorValue}", Position + new Vector2(0, baseY +
tinyFont.LineSpacing * 5), Color.Lime);
    spriteBatch.DrawString(tinyFont, $"Location:
{SelectedItem.EquipableLocation}", Position + new Vector2(0, baseY +
tinyFont.LineSpacing * 6), Color.Lime);
}

if (SelectedItem is IAmmunition)
{
    spriteBatch.DrawString(tinyFont, $"Quantity:
{((IAmmunition)SelectedItem).Quantity}", Position + new Vector2(0, baseY +
tinyFont.LineSpacing * 5), Color.Lime);
    spriteBatch.DrawString(tinyFont, $"Weapon: {string.Join('-
',((IAmmunition)SelectedItem).Weapons.ToArray())}", Position + new
Vector2(0, baseY + tinyFont.LineSpacing * 6), Color.Lime);
}

if (SelectedItem is IConsumable)
```

```
{
    spriteBatch.DrawString(tinyFont, $"Quantity:
{((IConsumable)SelectedItem).Quantity}", Position + new Vector2(0, baseY +
tinyFont.LineSpacing * 5), Color.Lime);
}
```

For example, with the sword selected we get the following:

Figure 24: Sword info

And with the shield:

Figure 25: Shield info

We are keeping track of the current selected item by setting the **`SelectedItem`** in the **`HandleInput`** method. When an arrow key is pressed, it's used to move the current item index along and change the current selected item.

Code Listing 102: Inventory navigation

```
if (input.IsNewKeyPress(Keys.Right, playerIndex, out pidx))
{
    int idx = Items.IndexOf(SelectedItem);
    idx = MathHelper.Min(++idx, Items.Count - 1);

    SelectedItem = Items[idx];
}

if (input.IsNewKeyPress(Keys.Left, playerIndex, out pidx))
{
    int idx = Items.IndexOf(SelectedItem);
    idx = MathHelper.Max(--idx, 0);

    SelectedItem = Items[idx];
}

if (input.IsNewKeyPress(Keys.Down, playerIndex, out pidx))
{
    int idx = Items.IndexOf(SelectedItem);
    idx = MathHelper.Min(idx + invSize, Items.Count - 1);
    SelectedItem = Items[idx];
```

```
}

if (input.IsNewKeyPress(Keys.Up, playerIndex, out pidx))
{
    int idx = Items.IndexOf(SelectedItem);
    idx = MathHelper.Max(idx - invSize, 0);
    SelectedItem = Items[idx];
}
```

We can also use the mouse to track or select the current item by either clicking on an item or the scroll buttons.

Code Listing 103: Inventory mouse navigation

```
foreach (ItemBase item in Items)
{
    if (input.MousePointerRect.Intersects(item.BoundsRectangle))
    {
        mouseOver = item;

        if (input.IsNewMouseButtonPressed())
            SelectedItem = item;

        break;
    }

}

if (input.MousePointerRect.Intersects(up.BoundsRectangle))
{
    up.Tint = Color.DarkCyan;
    if (input.IsNewMouseButtonPressed())
    {
        up.Tint = Color.Cyan;
        int idx = Items.IndexOf(SelectedItem);
        idx = MathHelper.Max(idx - invSize, 0);
        SelectedItem = Items[idx];
    }
}
else
    up.Tint = Color.White;

if (input.MousePointerRect.Intersects(down.BoundsRectangle))
{
    down.Tint = Color.DarkCyan;
    if (input.IsNewMouseButtonPressed())
    {
        down.Tint = Color.Cyan;
```

```
        int idx = Items.IndexOf(SelectedItem);
        idx = MathHelper.Min(idx + invSize, Items.Count - 1);
        SelectedItem = Items[idx];
    }
}
else
    down.Tint = Color.White;
```

PlayerInventory render

Let's now have a look at how the **`PlayerInvenory`** renders and handles input.

We start by drawing the background sprite and the sprite used to show what items are equipped, and where, on the avatar. We then render a slot for each equipable item slot.

Code Listing 104: Rendering equipment slots

```
Background.Position = Position;
WearingBG.Position = Position + new Vector2(8, titleSize.Y + 32);

Background.Draw(gameTime, spriteBatch);
WearingBG.Draw(gameTime, spriteBatch);

// Draw equipable slots...
// Head
DrawEquippedSlot(EquipableLocation.Head, gameTime, spriteBatch);

// Chest
DrawEquippedSlot(EquipableLocation.Chest, gameTime, spriteBatch);

// Left Arm
DrawEquippedSlot(EquipableLocation.Left_Arm, gameTime, spriteBatch);

// Left Hand
DrawEquippedSlot(EquipableLocation.Left_Hand, gameTime, spriteBatch);

// Right Hand
DrawEquippedSlot(EquipableLocation.Right_Hand, gameTime, spriteBatch);

// Right Arm
DrawEquippedSlot(EquipableLocation.Right_Arm, gameTime, spriteBatch);

// Abdomen
DrawEquippedSlot(EquipableLocation.Abdomen, gameTime, spriteBatch);

// Left Leg
DrawEquippedSlot(EquipableLocation.Left_Leg, gameTime, spriteBatch);
```

```
// Right Leg
DrawEquippedSlot(EquipableLocation.Right_Leg, gameTime, spriteBatch);

// Feet
DrawEquippedSlot(EquipableLocation.Feet, gameTime, spriteBatch);
```

We can then render the inventory system.

Code Listing 105: Render inventory

```
if (Inventory.IsShowing)
    Inventory.Draw(gameTime, spriteBatch);
```

We also render the equipable location and the name of the item if the mouse is over it.

Code Listing 106: Render equipped items and slots

```
foreach (EquipableLocation loc in Equipped.Keys)
{
    IInventoryItem item = Equipped[loc];
    if (item != null && item == mouseOverItem)
    {
        string iloc = $"{loc.ToString().Replace("_"," ")}: {item.Name}";

        Point s = smallFont.MeasureString(iloc).ToPoint();
        Point p = (item.Position + new Vector2(0, smallFont.LineSpacing *
2)).ToPoint();

        spriteBatch.Draw(SlotBox.spriteTexture, new Rectangle(p.X-4, p.Y-4,
s.X+8, s.Y+8), Color.Gold);
        spriteBatch.DrawString(smallFont, iloc, p.ToVector2(),
Color.White);
    }
}
```

We can now handle the user input for the **`PlayerInventory`**.

Code Listing 107: Handle input

```
public void HandleInput(GameTime gameTime,PlayerIndex? playerIndex,
InputState input)
{
    PlayerIndex pidx;
    Inventory.HandleInput(gameTime, playerIndex, input);

    // Drop/remove item
```

```
    if (Inventory.SelectedItem != null && input.IsNewKeyPress(Keys.D,
playerIndex, out pidx))
    {
        // Drop the selected item
        Drop(Inventory.SelectedItem);
    }

    // Equip item
    if (input.IsNewKeyPress(Keys.Enter, playerIndex, out pidx) &&
Inventory.SelectedItem != null)
    {
        EquipItem(Inventory.SelectedItem);
    }

    mouseOverItem = null;

    // Mouse over any equipped items?
    foreach (EquipableLocation location in Equipped.Keys)
    {
        if (Equipped[location] != null)
        {
            ItemBase item = (ItemBase)Equipped[location];

            if (input.MousePointerRect.Intersects(item.BoundsRectangle))
            {
                mouseOverItem = item;

                if (input.IsNewMouseButtonPressed())
                {
                    UnEquip(location);
                    break;
                }
            }
        }
    }
}
```

As you can see, we can drop the selected item when the D key is pressed, we can equip the selected item when the Enter key is pressed, and we can unequip an equipped item when we click on it.

What's next

We now have equipment we can use in our world. Let's look at how we might use some of these weapons we have created in combat!

Chapter 9 Combat

Introduction

When many people think of RPGs, they imagine a character in a suit of armor or mage's robes wandering around in a dungeon battling the various creatures that live there. Granted, combat is usually a big part of RPGs, but it isn't everything as we've already seen. However, it does happen to be where we are—so hacking and slashing and throwing fireballs it is.

Types of combat systems

When implementing a combat system, the first decision is the type. Usually, a combat system runs in real time, is turned-based, or may be a hybrid of the two (pause-able real-time, for example). Even a system that appears to be real-time may have hidden pieces that make it a more turned-based game. Most games don't let you swing a sword as quick as you can click the mouse, for example.

For our small sample game, we'll implement a system similar to some of the older Final Fantasy games. Combat takes place on its own screen, separate from the game world:

Figure 26: Final Fantasy IV combat

This is probably one of the easier types of combat to implement, as you don't have to account for the player doing something unexpected.

Initiating combat

Combat can be started in one of two ways: the player clicking on an enemy, or an enemy noticing the character and attacking. For our purposes, we'll just concentrate on the former. This means changing the existing code that just automatically killed an enemy and making it display the combat screen when the player clicks on an enemy.

Since we're not doing any kind of fog of war or line of sight, the player will see everything in the level (this is a change you'll probably want to implement in your game 😊). It wouldn't make sense to start combat with an enemy the character can't even see, but for our purposes we're not going to worry about that.

It's fairly easy to start combat:

Code Listing 108: Combat initialization

```
npcs = new List<EntityGameObject>();
npcs.AddRange(GameObject.LoadNPCs());

foreach (EntityGameObject obj in npcs)
{
    obj.Initialize(obj.GameSpriteFileName);
    obj.NPCClicked += NPCClicked;
    obj.CreatureClicked += CreatureClicked;
}

private void CreatureClicked(EntityEventArgs e)
{
    //Start combat
    List<Entity> opponents = new List<Entity>();
    opponents.Add(((Entity)npcs.Find(n => ((Entity)n.Entity).ID ==
e.ID).Entity));

    ScreenManager.AddScreen(new CombatScreen((Character)character.Entity,
opponents), ControllingPlayer);
}
```

We've removed killing the entity and replaced it with an event handler that will display the combat screen when the entity is clicked.

Our combat screen won't be up to the Final Fantasy standards, but it'll do for our purposes:

Figure 27: Combat screen

Once combat has started, the **`CombatManager`** takes over. The **`CombatManager`** communicates to the combat screen information such as which side's turn it is, the player or the enemy; when an entity is damaged or killed; and when the current state of the combat has changed.

Combat state

There are many states that combat goes through. While we won't use all of them, the following are some possibilities:

Code Listing 109: Combat turn states

```
public enum CombatTurnState
{
    DetermineInitiative,
    InitiativeDetermined,
    PlayerTurn,
    PlayerTurnDone,
    OpponentTurn,
    OpponentTurnDone,      // This is just one opponent in the list
    CharacterActionSelect,
    CharacterActionSelected,
    ResolveTurn,
    TurnResolved,
    RoundEnded,
    CombatEnded
}
```

The first thing that happens in combat is figuring out whose turn it is. Since our combat system is turn-based, each round of combat goes through two turns, the player and the enemy.

During the player's turn, a series of menus will be displayed based on the character.

After the player's turn, the **`CombatManager`** handles all the actions of the enemy.

After both sides have taken their turns, if both sides are still alive, the next round begins, going back to the **`DetermineInitiative`** state. This continues until all the entities on one side are dead.

Some of the states are just used to communicate back to the combat screen so the player knows what is going on if the current state doesn't require them to make a decision.

The **`CombatManager`** only has a few members and methods for the relatively simple combat system we're implementing. We'll take a look at them now.

Code Listing 110: Combat manager class

```
public class CombatManager
{
    private Character character;
    private List<Entity> opponents;

    private CombatTurnState state;

    public delegate void
CombatManagerStateChangedEventHandler(CombatManagerStateChangedEventArgs
e);
    public event CombatManagerStateChangedEventHandler StateChanged;

    private Dictionary<int, int> initiativeOrder;   //Key = indices of
opponents and character, character is -1
                                                    //Value = initiative
roll

    private int curInitiative;  //Value from 1-100, decremented until a
value in dictionary is found

    private int curEntity;

    private List<int> keys;
}
```

Since a player can fight multiple enemies during combat, we have a list of **`Entity`** objects to hold them.

The **`State`** member is the current state in the possible states we just looked at.

When the state changes, it's communicated to the combat screen through the **`StateChanged`** event. We'll take a look shortly at how this is hooked up. It's no different than the other event handlers we've implemented, but what happens will vary with the state that's passed through.

For each turn, the entities involved in the combat have a random number from 1–100 rolled for them to indicate when they'll be able to perform an action during that turn. We store those rolls in the **`initiativeOrder`** member using the entity's ID as a key. Since the character doesn't have a distinct ID, we use -1 for the character.

The **`curInitiative`** member holds the value of the current number from 1–100. We count down from 100, since the highest number acts first during the turn, until we find an entity with that value.

The **`curOpponent`** is the index in the list of opponents the player is fighting.

We have a handful of methods in the **`CombatManager`** to go along with the members.

Code Listing 111: Setting the initiative order

```
public void SetInitiativeOrder()
{
    initiativeOrder.Clear();

    initiativeOrder.Add(-1, GlobalFunctions.GetRandomNumber(DieType.d100));

    for(int i = 0; i < opponents.Count; i++)
        initiativeOrder.Add(i,
GlobalFunctions.GetRandomNumber(DieType.d100));

    curInitiative = 101;

    state = CombatTurnState.InitiativeDetermined;
    StateChanged?.Invoke(new CombatManagerStateChangedEventArgs() {
NewState = state
    });
}
```

Setting the initiative is just a matter or calling our method to get a random number for each entity in the combat. We then let the combat screen know this has been finished. The combat screen will then ask for the highest value the entities have to start the first turn:

Code Listing 112: Getting the next initiative key

```
public List<int> GetNextInitiativeKey()
{
    while (true)
    {
        curInitiative--;

        if (curInitiative == 0)
            break;

        foreach(KeyValuePair<int, int> kvp in initiativeOrder)
        {
            if (kvp.Value == curInitiative)
                keys.Add(kvp.Key);
        }

        if (keys.Count > 0)
            break;
    }

    if (keys.Count == 0)
    {
        state = CombatTurnState.RoundEnded;
```

```
        StateChanged?.Invoke(new CombatManagerStateChangedEventArgs() {
NewState = state });
    }
    else
    {
        List<int> temp = new List<int>();

        //Order by dex
        keys.Sort(delegate(int a, int b)
        {
            return a.CompareTo(b);
        });

        if (keys[0] == -1)
            state = CombatTurnState.PlayerTurn;
        else
            state = CombatTurnState.OpponentTurn;

        curEntity = 0;

        StateChanged?.Invoke(new CombatManagerStateChangedEventArgs() {
NewState = state });
    }

    return keys;
}
```

Since multiple entities may have the same initiative number, we hold them in a list and sort them based on the entity's dexterity stat, figuring that a higher dexterity would allow an entity to strike quicker.

We set the current state appropriately, depending on whether the character is the first entity in that list. We then let the combat screen know.

If it's the character's turn, the combat manager waits for the character to select their action; otherwise, we call a method that lets the enemy attack:

Code Listing 113: Processing the opponent's action

```
private void DoOpponentAction()
{
    if (!opponents[curEntity].IsDead())
    {
        //This should eventually get the equipped weapon
        int damage =
((Weapon)Globals.Items[opponents[curEntity].Inventory[0].ID]).GetDamage();
        short roll = GlobalFunctions.GetRandomNumber(DieType.d100);

        if (roll >= 100)
        {
            character.Damage(damage);

            if(character.IsDead())
                state = CombatTurnState.CombatEnded;
            else
                state = CombatTurnState.OpponentTurnDone;
        }
        else
            state = CombatTurnState.OpponentTurnDone;
    }
    else
        state = CombatTurnState.OpponentTurnDone;

    StateChanged?.Invoke(new CombatManagerStateChangedEventArgs() {
NewState = state });
}
```

Since we're not removing enemies from the list once they're dead, we need to do a check before allowing them to perform an action. (It's hard for dead people to do things, unless they're a zombie—and this isn't that kind of game.)

We're assuming the first item in the entity's inventory is their weapon. This should be changed to verify the entity actually has a weapon equipped (or has a natural weapon like teeth and claws, which we don't implement in this abbreviated RPG).

We then do a random roll and, if the roll is 100 or greater, damage the character. Obviously, this isn't going to happen a lot, so something else should be added to the roll. This would be taking into account the character's skill with the weapon, any enhancements on the weapon like spells that make the weapon easier to use, and so on. We have everything in place to do that in the various classes; it just needs to be implemented. This is another thing that is difficult to cover in a book like this but is easy enough to actually do. You just need to decide how to implement these pieces as there are a number of ways to do it.

If the character has been killed, combat is over; otherwise, we let the combat screen know that the opponent is done.

During the character's turn, we're only handling casting an offensive spell. The logic for any other offensive attack would be similar.

Code Listing 114: Code to handle casting a spell

```
public bool CastSpell(int targetIndex, int spellIndex, out int amount)
{
    short roll = GlobalFunctions.GetRandomNumber(DieType.d100);
    bool ret = false;

    Spell spell = Globals.Spells[character.GetSpellByIndex(spellIndex).ID];
    SpellType type = spell.Type;

    if (roll >= 100)
    {
        amount = spell.DamageAmount;

        switch (type)
        {
            case SpellType.Defensive:
                {

                    break;
                }
            case SpellType.NonCombat:
                {

                    break;
                }

            case SpellType.Offensive:
                {
                    opponents[targetIndex].Damage(amount,
DamageType.Magical);

                    CheckForCombatOver();

                    break;
                }
        }

        ret = true;
    }
    else
    {
        amount = 0;
    }

    if (state != CombatTurnState.CombatEnded)
    {
```

```
        state = CombatTurnState.PlayerTurnDone;

        StateChanged?.Invoke(new CombatManagerStateChangedEventArgs() {
NewState = state });
    }

    return ret;
}
```

The character's attack is handled similarly to the enemy's—roll a die; if it's 100 or greater, damage the enemy, then check to see if the character has died. If so, as with the enemy attacking, the combat is over. Otherwise, we let the combat screen know the player's turn is over. We notify the combat screen either way.

When any entity has finished and combat is not over, we get the next entity in the current initiative order:

Code Listing 115: Figuring out the next entity in the combat turn

```
public void GetNextKey()
{
    curEntity++;

    if (curEntity == keys.Count)
    {
        GetNextInitiativeKeys();
    }
    else
    {
        if (keys[curEntity] == -1)
        {
            state = CombatTurnState.PlayerTurn;
        }
        else
        {
            state = CombatTurnState.OpponentTurn;
        }

        StateChanged?.Invoke(new CombatManagerStateChangedEventArgs() {
NewState = state });

        if (state == CombatTurnState.OpponentTurn)
        {
            DoOpponentAction();
        }
    }
}
```

If all the entities in the current initiative value have finished, we call the method that generates the random number for the round. Otherwise, we get the next entity, figure out if it's the player or not, and set the appropriate state. If it's the enemy's turn, we call the method to let the enemy attack.

What's next

While the combat system does the basic job, it's not perfect. There are a lot of elements that could be added depending on your specific game. There are also some optimizations that can be made, such as quickly overwriting displayed messages with the next message, giving the player the ability to change weapons, or showing the hit points for the combatants.

As far as completing the game, when enemies are defeated and quests are completed, the character may have the ability to become more powerful. This may be done in a number of ways. We'll look at how we'll be doing it in the next chapter.

Chapter 10 Character Development

Level up!

There are a million and one systems in traditional RPGs for advancing your character. These vary from experience and level systems where characters accumulate points until they reach a threshold to attain the next level, to skill-based systems where during the adventure, if a skill is used successfully, the player gets a chance to improve that skill.

We are going to have a look at some of these systems and try to help you decide how to implement character advancement in your game based on the topics covered in earlier chapters.

XP and levels

As your character gains experience points (XP) during the course of their adventures, and once they reach a threshold of XP, they then ascend to the next level (after possibly paying a training fee) and attain better stats. They'll have skills and spells up to their new level available to them, and as leveling-up systems go, it works—but with the drawback of having all those skill tables and advancements already planned out.

While I think it's a great way to have character advancement, I think it's limited. With this method, you are sort of left with a ceiling where you end up having to produce add-ons to your game to cater for higher-level characters.

Your RPG may well be story-based, in which case the notion of high-level characters is a moot one. In this scenario, you would want the player to develop their character to a point that enables them to tackle the next part of the story. This then means that the advancement of the character is actually used to advance the story.

Your story may have a number of gateways based on the character type. For example, in order for a player using a warrior or fighter to be able to find the entrance to the cave, they need to have gained enough experience points to reach level 2. Or as a mage, they typically are not as resilient as fighters in the early stages of their development, so maybe they need to be level 4 where they have access to stronger attacking spells in order for them to tackle the denizens of the cave. A cleric may need to be level 3, and so on.

XP and careers

With Warhammer Fantasy RPG 1st Edition, a career-based system provided the player with a list of basic careers they could enter into rather than rigid classes like D&D and other RPGs.

Within the player's chosen career, as they gained experience, they spent points advancing their stats and buying skills related to that career, paying more for advancements outside of it. Once

all the stat and skill advancements are taken, the player could then choose from a list of careers to take up next and start advancing their character again. The player could even decide to choose a totally new career path (at extra cost) and try their hand at something else. This could be another basic career, or it could lead to a more advanced career.

I really like this sort of character advancement. While still quite rigid like the XP and levels, it feels a bit more realistic. In the real world, we all change jobs and roles throughout our lives, and this sort of system seems to be quite reflective of life itself—though with added magic spells and sharp swords.

A character's career development could be something along the lines of starting out as a Rat Catcher in the city where they have a number of adventures and reach the end of the advancements this career provides, accumulating increased initiative, the secret sign of the Thieves Guild, the ability to set simple traps, and so on in the process. The career exits for a Rat Catcher could be Stable Hand, City Watch, Pick Pocket, or Charlatan. Say the player chooses City Watch. This career comes with chain mail and a short sword, and the player's knowledge of the city and the Thieves Guild sign from their days as a Rat Catcher may prove useful.

Again, this relies on you predefining the careers and the career trees for the career exits. It's a lot of work up front, but I think it also adds a bit more lore to your game and pulls your character and player more into the world you have created.

Skills and usage

With Call of Cthulhu and Traveller (the little black books, or LBBs as they are sometimes known), the enhancement of stats tends to come during adventures or from acquiring drugs and/or equipment, and skills are increased if used during play. At the end of the session, scenario, or campaign, there is a chance that those skills used during play are improved.

This way of leveling up a character lends itself quite well to video RPGs. With this method, we don't need to hold a host of tables as the player levels up, or have special characteristics against a given item, spell, or weapon that only a given few can use or are excluded from using. It is a more open and natural system and allows for unstifled character growth.

In this sort of system, there does tend to be a penalty for players as they age or gain experience. For example, as a player ages, they may start to get penalties against certain statistics, and some skills themselves may have detrimental effects on the character as they advance. For example, in Call of Cthulhu, as your Mythos (knowledge of the esoteric aliens in the game) increases, your maximum sanity is reduced proportionally.

One possibility

Since we already have skill selection and stat increases along with a player class, one possibility for advancement would be to combine experience and character levels with skill

selection based on points earned in each level and increasing skills through having the player make the selection or based on the character's class.

Doing this would mean some work determining the experience needed for each level, along with points given for those levels.

Experience calculation types

Experience requirements usually scale in some fashion. Here are a couple types of growth:

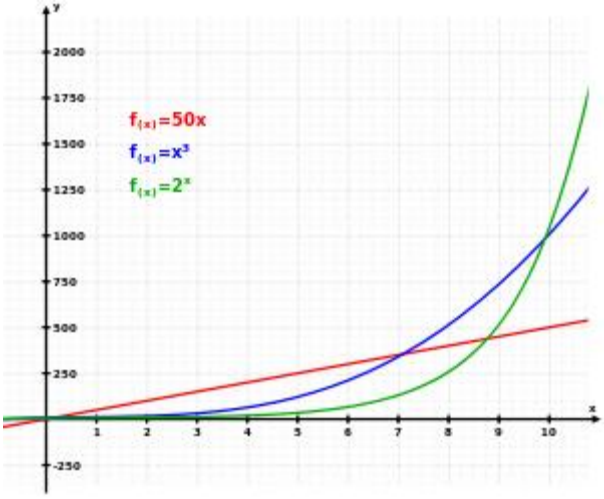


Figure 28: Experience growth

The red line is linear, where the level is multiplied by some constant. Each level is about the same difficulty to attain. This probably isn't the best solution. Level 50 should be much more difficult to reach than level 2.

The blue line is cubic growth, where the level is cubed to find the experience needed. This provides a somewhat gentle curve in difficulty.

The green line is exponential growth, where the level becomes the power that's used with a constant. This starts out with low difficulty, but quickly becomes increasingly difficult.

You could even mix the different curve formulas. For example, you could do $x^3 + 50x$. Doing this allows you to tailor the difficulty at the expense of being a bit harder to understand.

Skill and stats

Once you have the levels in place, you can determine how skills and stats can increase.

For skills, you can give the player a set number of points, say 10, to spend at each level on new skills or to increase existing ones. As we already have a screen in place for this, this is minimal

work. You'll have to do a bit of adjusting so the screen knows that you're using it for level advancement and not creating a new character, but this shouldn't be difficult.

Stats require a bit more finesse, as the character can quickly become over-powered if you allow them to increase stats every level. Staggering stat increases is one possible fix. Stats could increase every other level, or you can have one out of several increase each level. A fighter, for example, could alternate between strength and constitution every level or every other level. You can also cap stat increases, only allowing the character to increase stats by a certain amount. No character should "realistically" (in quotes since this is a fantasy game after all) have 100 for every stat by the end of the game.

Spells

Magic-using characters should be able to learn new spells as they advance in level. This is something you can control by simply using the existing skill-buying structure or allowing the player to select one or more new spells every level or every few levels. Again, you can tailor this as you like.

Player knowledge of system

However you implement your system, it may be a good idea to provide some or all of the details on how a character advances to the player up front to allow them to choose based on how they want to play.

For example, if you make magic-using characters advance more slowly in relative power than fighters, making the player aware of this up front could prevent frustration if their character doesn't seem to be getting as powerful as quickly as they expect.

What's next

We now have some ideas for character advancement, but so far we have not covered an important element of the game—the audio.

Chapter 11 Audio

While this is probably going to be the shortest chapter in this book, it is by no means the least important, nor is the subject matter even remotely light. Audio in games is an incredibly important element, as it is in stage and film—especially in modern games.

Back when computer games were a relatively new phenomenon, we didn't really have access to the best sound chips to create audio in our games. We pretty much had the humble beep, but we could change the duration and pitch of that beep, and this led to some really creative audio in some of the early games.

A few ZX Spectrum games spring to mind that had some great lo-fi music:

- Saboteur by Durell (ref)
- Airwolf by Elite Systems Ltd (ref)

A few years later, the chipsets moved on quite a bit with machines like the Atari ST having a dedicated Yamaha sound chip for audio, and again, a few great games and their audio spring to mind:

- Paradroid 90 by Graftgold Ltd (ref)
- Captain Blood by Exxos, whose sound was done by the famous Jean-Michel Jarre (ref)
- Speedball by The Bitmap Bros (ref)
- Xenon II by The Bitmap Bros (ref)

The amazing demo scene at the time was full of incredible audio demos, too (ref , ref2 ref3, ref4, ref5).

Then, as the console became popular, we had awesome audio for games like the following:

- Wipeout by Psygnosis (ref)
- Gran Turismo by Polyphony Digital & Cyberhead (ref)
- StarFox by Nintendo (ref)

In more modern times, the sound quality really is taken to the next level with games such as Skyrim by Bethesda (ref), Red Dead Redemption by Rockstar (ref), and GTA V by Rockstar (ref).

Ambient sound

Audio in a role-playing game, as with most games, is used to great effect to convey the emotion of the player in a scene or area. Ambient sound alone can really set the scene and emotion of the current situation within the game. We can use it to convey impending physical doom, the tranquility of a magical clearing lit by shafts of sunlight, or the cold sparsity of the vacuum in space.

Game audio

Knowing how important audio is in our game, how do we create or get the audio we need?

Well, you could invest in the software to make your own audio for your games, but if you are anything like us, your budget might not go that far. There are free applications out there that you can use, though.

CakeWalk is free to download and use if you have a BandLab account. If you are after that 8-bit feel for your audio, then check out BeepBox; it's one of many 8-bit online resources you can use to create music.

If, like me, you don't have a musical ear, then you could also search online for license-free audio. There are a number of sites out there, but we came across an awesome member of the MonoGame community going by the name of SoundImage who is posting a wealth of music and, as their name suggests, images for MonoGame developers to use in their projects. You can check out their community post here, or go directly to their site here.

Audio

As we know from all the games we have played, audio in our game is important. We want to set the mood for our game right at the start. A nice way to do this (after your splash screens) is to have some appropriate music.

We have put the **`AudioManager`** from the **`VoidEngineLight`** library into the **`ScreenManager`** class, so we can control pretty much all our audio from here. We have also added a **`BackgroundSoundAsset`** to both the **`ScreenManager`** class and the **`GameScreen`** class.

Code Listing 116: AudioManager

```
public IAudioManager audioManager
{
    get
    {
        return Game.Services.GetService<IAudioManager>();
    }
}
```

In the **`Update`** function, we can now play music based on what has been set. If a screen does not have a background asset set, then it is assumed that the **`ScreenManager`** background sound is to be used.

Code Listing 117: ScreenManager update

```
string musicAsset = BackgroundSongAsset;

if (!string.IsNullOrEmpty(screen.BackgroundSongAsset))
    musicAsset = screen.BackgroundSongAsset;

if (!string.IsNullOrEmpty(musicAsset))
{
    if (!audioManager.IsMusicPlaying)
        audioManager.PlaySong(musicAsset, 1, true);
    else
    {
        if (musicAsset != audioManager.CurrentSongAsset)
        {
            // Should really fade out and back in...
            audioManager.StopMusic();
        }
    }
}
```

We now have a mechanism for playing the music we want per screen, and as you can see from the code, we can switch to another track by simply setting the current screen's **`BackgroundSongAsset`**.

Sound effects (SFX)

Now, within our menu, we will want to add some audio to the menu options. Again, we want to find some suitable sounds for moving from option to option. I have chosen a short UI sound. All we need to do now is add a call to our audio manager in the menu navigation code inside the **`MenuScreen`** class.

Code Listing 118: MenuScreen HandleInput

```
if (_menuUp.Occurred(input, ControllingPlayer, out playerIndex))
{
    ScreenManager.audioManager.PlaySFX("Audio/SFX/ui_select_2");
    _selectedEntry--;

    if (_selectedEntry < 0)
        _selectedEntry = _menuEntries.Count - 1;
}

if (_menuDown.Occurred(input, ControllingPlayer, out playerIndex))
{
    ScreenManager.audioManager.PlaySFX("Audio/SFX/ui_select_2");
    _selectedEntry++;

    if (_selectedEntry >= _menuEntries.Count)
        _selectedEntry = 0;
}

if (_menuSelect.Occurred(input, ControllingPlayer, out playerIndex))
{
    ScreenManager.audioManager.PlaySFX("Audio/SFX/ui_accept_2");
    OnSelectEntry(_selectedEntry, playerIndex);
}
else if (_menuCancel.Occurred(input, ControllingPlayer, out playerIndex))
{
    ScreenManager.audioManager.PlaySFX("Audio/SFX/ui_error_2");
    OnCancel(playerIndex);
}
```

As you can see, we will play a sound effect file each time we change the menu option, and another when we select an option or cancel.

We can do this pretty much anywhere in our game where we want to play a sound effect, such as when we hit an NPC, when money changes hands, or when an item is equipped.

Volume control

In the Options menu, there is also an Audio Options screen. From here we can set the master volume; this is the overall volume control for both the music and the SFX sounds. We can also set the music volume independently of the master and SFX volume, as we can with the SFX.

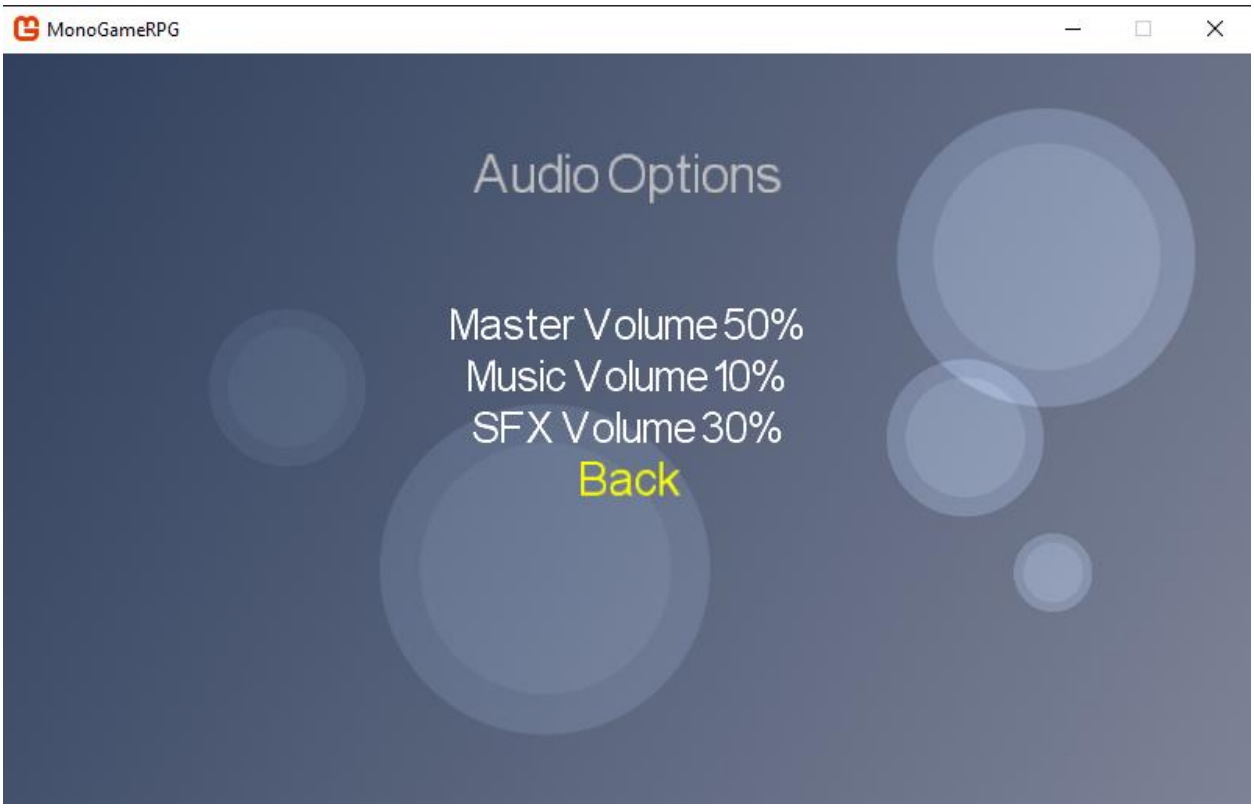


Figure 29: Audio options

When you leave this screen, the data is saved to file so the next time you load the game your audio settings will remain. This is done by simply saving the three volume values to a JSON file. Both the save and load of the audio settings are done from the **`ScreenManager`** class.

Code Listing 119: Load and save audio settings

```
public void SaveAudioSettings()
{
    AudioSettings settings = new AudioSettings(audioManager);

    string json = JsonConvert.SerializeObject(settings);

    File.WriteAllText("AudioSettings.json", json);
}

public void LoadAudioSettings()
{
    if (File.Exists("AudioSettings.json"))
    {
        string json = File.ReadAllText("AudioSettings.json");
        AudioSettings settings =
JsonConvert.DeserializeObject<AudioSettings>(json);

        settings.SetAudioManager(audioManager);
    }
    else
        SaveAudioSettings();
}
```

We are using the ever-useful NuGet package NewtonSoft.Json to do the serialization. The **`AudioSettings`** class is a simple internal class to the **`ScreenManager`** class that holds the three volumes and helps us get them from the audio manager and set them back again.

Code Listing 120: AudioSettings class

```
internal class AudioSettings
{
    public float MasterVolume { get; set; }
    public float MusicVolume { get; set; }
    public float SFXVolume { get; set; }

    public AudioSettings(IAudioManager audioManager = null)
    {
        if (audioManager != null)
        {
            MasterVolume = audioManager.MasterVolume;
            MusicVolume = audioManager.MusicVolume;
            SFXVolume = audioManager.SFXVolume;
        }
    }

    public void SetAudioManager(IAudioManager audioManager)
    {
        audioManager.MasterVolume = MasterVolume;
        audioManager.MusicVolume = MusicVolume;
        audioManager.SFXVolume = SFXVolume;
    }
}
```

The loading of the audio is done in the **`ScreenManager LoadContent()`** method.

What's next

Well, that is now up to you…

Summary

By this point, you have all the tools at your disposal to create any RPG you can envision. Be aware, any RPG is a lot of work. Up-front planning and realizing this fact may save you from becoming frustrated at how long it will take you to complete your masterpiece.

Have fun with the experience. Test often and get feedback from people you know who enjoy RPGs. This can go a long way to making your game enjoyable for every player.