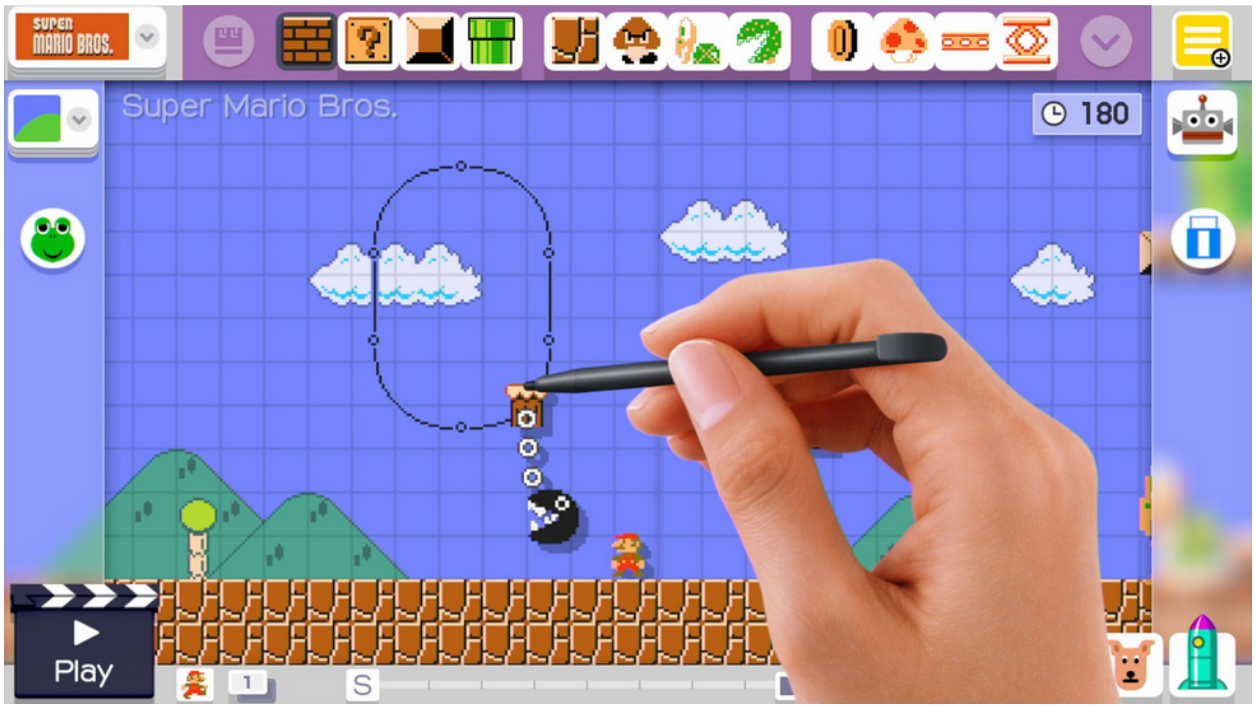
# Let’s Talk Games

*CS4730:Game Design*




Timur Anvar

March 2023

# INTRODUCTION

Welcome to "Let's Talk Games," a tutorial series designed to explore the fascinating world of game design and development. Whether you're an aspiring game designer, hobbyist game developer, or simply curious about the creative process behind video games, this series is tailored to meet your needs. In this tutorial, we will cover the fundamentals of game development, including topics such as graphics, organization, and prototyping. Each chapter will provide valuable insights into these aspects, along with practical tips and examples, to help you create engaging and exciting games.

Whether you're looking to create the next blockbuster or simply want to enhance your skills as a game developer, let's dive into the thrilling world of game design together!

## Let’s Talk Resolution

Resolution is an important aspect of game development that can greatly impact the player's experience. It refers to the number of pixels on the screen and determines the level of detail in the game's visuals. In this chapter, we'll discuss the basics of resolution and how to choose the right resolution for your game.

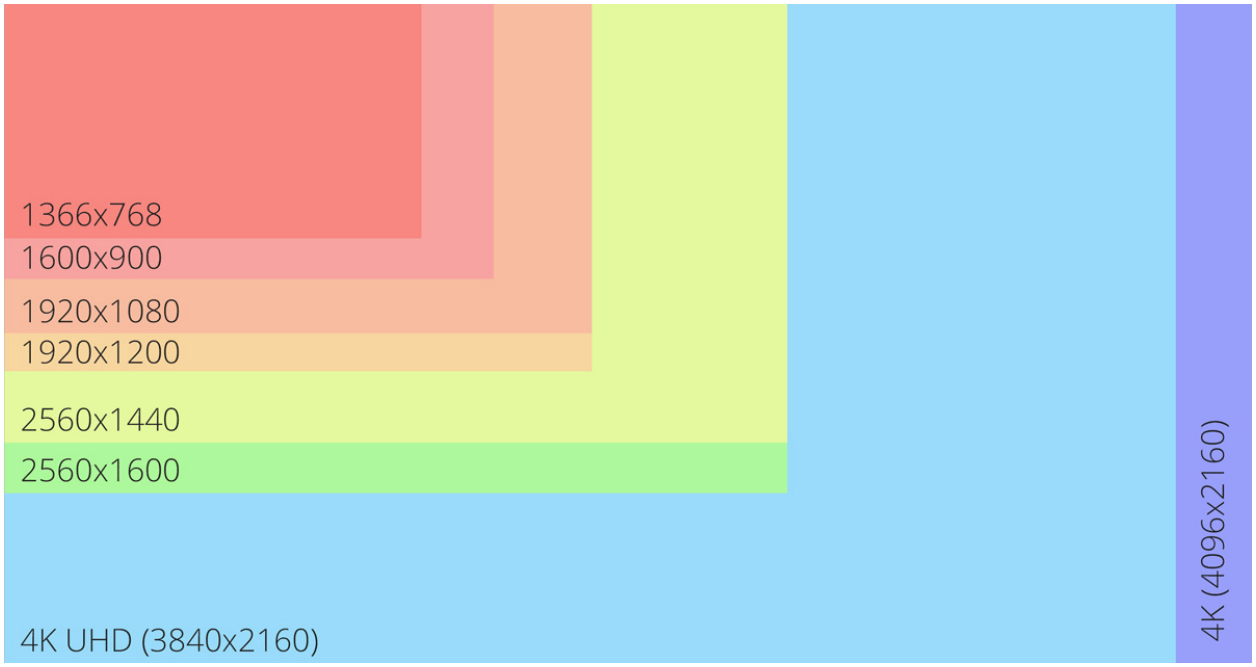
First, it's important to understand the different types of resolutions. There are two main types: standard definition (SD) and high definition (HD). SD resolutions typically have a 4:3 aspect ratio, while HD resolutions typically have a 16:9 aspect ratio. When choosing a resolution for your game, you'll need to consider your target audience and the devices they're using to play your game. For example, the most common aspect ratio for PC gaming is 16:9.

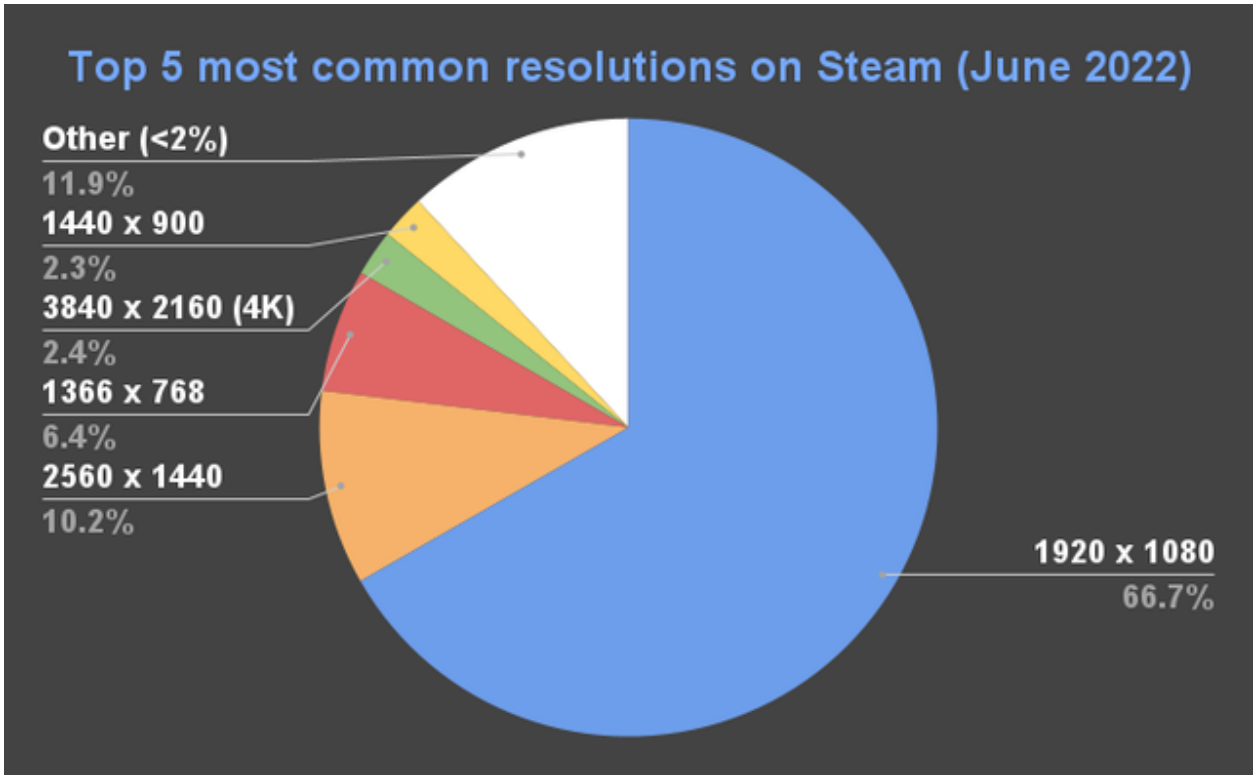
Another important consideration is how to handle different screen sizes and resolutions across different devices. Mobile devices, for example, often have a wide range of screen resolutions and aspect ratios, so it's important to ensure that your game is optimized to run on as many devices as possible. This can involve implementing responsive design techniques, using scalable vector graphics, or providing different versions of your game for different devices.

Finally, it's important to consider the performance impact of different resolutions. Higher resolutions require more processing power and can impact the game's framerate, so it's important to optimize your game's rendering pipeline and graphics settings to ensure a smooth experience for players.

In this class, we will be creating a desktop game, and our target aspect ratio should be 16:9. To make the most of our time throughout the semester, we should create a game in the lowest possible resolution and scale it up to our preferred resolution. To ensure that the visuals remain sharp and clear when scaling up, we should use integer multiplication. This will help to avoid any distortion or blurriness that can occur when using non-integer scaling factors. In terms of a base resolution, a 640x360 resolution may be a good starting point for a 16:9 game. You could also choose 320x180 or 1240x720, but I would not recommend going lower or higher.

## Let’s Talk Resolution



## Let’s Talk Graphics

Above, we discussed the importance of the resolution of the game display, but it's also essential to consider the resolution of the graphic assets. The resolution of the assets significantly affects the overall visual quality of the game. Using low-resolution assets results in a pixelated and retro appearance, while high-resolution assets create a more modern and polished look. However, high-resolution assets tend to increase the game's file size, which can impact loading times and performance.

When it comes to creating desktop games, using 16x16 or 32x32 tilesets can strike a good balance between visual quality and performance. These resolutions are commonly used in retro-style games, and they allow for a good level of detail without sacrificing too much in terms of file size or processing power. Moreover, using these resolutions makes it easier to find free or low-cost assets and modify them, even if you have no art design background. If you need to create your assets, using these resolutions makes your task easier as they are simpler to create and can maintain the overall consistency of the game's aesthetic. And if you're willing to spend $20 to explore the world of pixel art, I recommend checking out Aseprite. It's user-friendly, easy to learn, and can sometimes be found on sale.

Here are some of the free 16x16 and 32x32 assets you can find useful:

1. Full Platformer: https://pixelfrog-assets.itch.io/pixel-adventure-1
2. Part 2 of the above: https://pixelfrog-assets.itch.io/pixel-adventure-2
3. Full Platformer, inside: https://pixelfrog-assets.itch.io/kings-and-pigs
4. Platformer, pirates: https://pixelfrog-assets.itch.io/treasure-hunters
5. Farm simulator: https://cupnooble.itch.io/sprout-lands-asset-pack
6. Farm, free for non-commercial use: https://shubibubi.itch.io/cozy-farm
7. Ninja RPG: https://pixel-boy.itch.io/ninja-adventure-asset-pack
8. GUI: https://mounirtohami.itch.io/pixel-art-gui-elements
9. Icons: https://cheekyinkling.itch.io/shikashis-fantasy-icons-pack
10. Knight: https://aamatniekss.itch.io/fantasy-knight-free-pixelart-animated-character
11. Archer: https://oco.itch.io/medieval-fantasy-character-pack-4

## Let’s talk about Complexity.

Developing a game is a challenging and time-consuming task that often involves unexpected obstacles and hurdles. One aspect that people sometimes overlook when choosing a game idea is the complexity involved in game development. It's always harder than you think it is. In this chapter, we'll discuss some practical tips for managing complexity and making the game development process more manageable.

1. Create a Plan and Milestones

   The first step in managing game complexity is to create a plan and milestones. Before diving into development, take some time to brainstorm your game idea and create a detailed plan outlining the game mechanics, levels, and overall design. Break down the development process into smaller, more manageable tasks and set achievable milestones that will help keep you on track.

2. Keep It Simple

   Every motion and action in a game increases complexity. For each action, you will need new animation and logic, which can quickly spiral out of control if you're not careful. To manage this, take your game idea and undress it to the bare minimum. Implement only the core mechanics needed to make the game playable and enjoyable, and then polish them until they're perfect.

3. Scale Up (Or Down).

   Once you have a solid foundation, you can begin to scale up your game's complexity. However, it's important to do this gradually and incrementally, adding new features one at a time and testing each one thoroughly before moving on to the next. Similarly, if you find that your game idea is too complex to manage, consider scaling it down by removing some of the motions/actions or simplifying the mechanics.

Managing game complexity is a crucial part of game development. By following the above, you can create a great game that is both fun to play and manageable to develop.

## Let’s Talk Complexity, Examples

1. Dodge Game.

   Begin with a player that moves right or left trying to avoid falling objects. Once it's done, introduce a new falling object that the player should collect. Now, add a happiness indicator that decreases every time a collectible is missed. Once this is done, introduce another object that falls less frequently but provides a buff. In this way, you started with a simple game and slowly built your way up.

2. Infinite platformer.

   A great starting point could be Google's T-Rex game, and you can gradually add complexity from like in the example above. Another approach is to experiment with a different game mechanic where the player moves vertically, either from bottom to top or top to bottom. This approach can introduce new challenges and gameplay elements, such as obstacles that move vertically or horizontally, power-ups that affect vertical movement, enemies that attack from above, below, or sides, or even environmental hazards like changing wind patterns.

3. Finite Platformer.

   Start by creating a level that consists only of regular obstacles. Then, slowly introduce new objects and mechanics one by one.

In conclusion, managing complexity is an essential part of game development. By breaking down your game idea into manageable tasks, focusing on the core mechanics, and scaling up gradually, you can create a well-polished game that engages players without overwhelming them.

## Let’s Talk Organization

As you embark on your game development journey, it is essential to understand the importance of project organization. This is particularly crucial when working in a team setting. Here are some tips:

1. Code Readability:

   One of the fundamental principles of writing code is to ensure that it is readable and understandable by other programmers. Therefore, you should avoid using shorthand or cryptic variable names that could confuse other developers working on the project.

2. Good Project Structure:

   To keep your project well-organized, you should divide it into different directories and subdirectories. For instance, you can use folders to store your game's assets, scripts, and resources. Besides, make sure to use appropriate naming conventions that are easy to understand and follow.

3. Avoid Writing Everything in a Single File:

   It's important to avoid writing everything in a single file, as it makes it difficult to keep track of things and divide work among team members. To simplify the development process, consider breaking your code into separate files based on their functionality.

4. OOP or ECS:

   When developing a game, it's essential to choose a programming paradigm that suits your project's needs. You can either use Object-Oriented Programming (OOP) or Entity Component System (ECS) to structure your code. Both paradigms have their advantages and disadvantages, so choose the one that works best for your project.

5. Create a Loader Class:

   To load all the game assets and resources, consider creating a loader class. This class will make it easy to manage all the game resources in one place, making it easier to modify and maintain your game assets.

6. Create a Game Constant Class:

   In every game, some constants remain the same throughout the game's life cycle. For example, screen width, screen height, gravity, and so on. It's a good practice to create a separate class to store all these constants for easy access and modification.

In summary, organizing your game development project is essential to ensure that you develop high-quality games that are easy to read, maintain, and modify.

## Let's Talk Organization, Loader Class

Below is an example of a Loader class:

```
using System.Collections.Generic;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace MyGame
{
//Constructor
    public static class Loader
    {
        private static Dictionary<string, Texture2D> textures = new
Dictionary<string, Texture2D>();
        public static void LoadContent(ContentManager content)
        {
        // Load textures for player
            textures.Add("player_idle",
content.Load<Texture2D>("Player/Frog/Idle"));
            textures.Add("player_idle_left",
content.Load<Texture2D>("Player/Frog/Idle"));
            textures.Add("player_right",
content.Load<Texture2D>("Player/Frog/Right"));
            textures.Add("player_left",
content.Load<Texture2D>("Player/Frog/Left"));
            textures.Add("player_jump",
content.Load<Texture2D>("Player/Frog/Jump"));
            textures.Add("player_double_jump",
content.Load<Texture2D>("Player/Frog/Double Jump"));
            textures.Add("player_fall",
content.Load<Texture2D>("Player/Frog/Fall"));
        // Load textures for game
```

```
            textures.Add("background_blue",
content.Load<Texture2D>("Background/Blue"));
        //Load Obstacles for game
            textures.Add("obstacle_ground",
content.Load<Texture2D>("Obstacles/Ground"));
        }
    //Retrieve a texture by its unique string name
        public static Texture2D GetTexture(string textureName)
        {
            if (textures.ContainsKey(textureName))
            {
                return textures[textureName];
            }
            return null;
        }
    }
}
```

In Game.cs you will need to load content using the Loader class above:

```
protected override void LoadContent()
    {
        _spriteBatch = new SpriteBatch(GraphicsDevice);
        // TODO: use this.Content to load your game content here
        Loader.LoadContent(Content);
    }
```

This class maps every sprite to a unique string. This way, you can always retrieve a texture by its unique key whenever necessary:

```
Texture2D texture = Loader.GetTexture(sprite);
```

## Let’s Talk Organization, ECS

Entity-Component-System (ECS) is a design pattern used in game development to organize game objects and their behaviors. It separates the game objects (entities) from their behaviors (components) and how those behaviors are processed (systems). The Entity is the core of the ECS design pattern. It represents a game object in the game world and is essentially just an ID. Entities do not contain any data or behavior themselves but rather are containers for components.

Components define the properties or behaviors of an entity. They are small and reusable units that contain data and behavior that can be mixed and matched to create different game objects. General examples of components are the Transform component (position, speed), Input Component (user's input), and Animated Component (spritesheets and frames). Components should be self-contained and ideally should not reference or interact with other components directly.

Systems are responsible for processing components and implementing behavior in the game world. They run on each frame or game tick and operate on all entities that have the required components. For example, an AnimationSystem might update all entities that have a SpriteComponent and an AnimationComponent. Systems should be completely unaware of the existence of entities, instead relying on the presence of the required components to operate.

The benefits of the ECS design pattern are numerous. It allows for the easy creation of new entities and behaviors by mixing and matching components, promotes code reuse through modular components, and allows for easy optimization by only updating the necessary components in a system. Additionally, it can make code more maintainable and flexible, making it easier to add new features or modify existing ones without affecting other parts of the codebase.

## Let's Talk Organization, ECS examples

Simple Component class:

```
using Microsoft.Xna.Framework;
namespace MyGame
{
    public abstract class Component
    {
        protected Entity entity;
        public virtual  void Update(GameTime gameTime)
        {
        }
        public virtual void OnDestroy()
        {
        }
        public void SetEntity(Entity entity)
        {
            this.entity = entity;
        }
    }
}
```

Note that I did not include a Draw method. This is just a very simple example, but you can modify it to fit your needs. Personally, I prefer to not have overloads for Drawing methods at the beginning of the project.

Simple Entity Class:

```
using Microsoft.Xna.Framework;
using System.Collections.Generic;
namespace MyGame
{
    public class Entity
    {
```

```
    private List<Component> components;
//Constructor
    public Entity()
    {
        components = new List<Component>();
    }
//Update
    public virtual void Update(GameTime gameTime)
    {
        foreach (Component component in components)
        {
            component.Update(gameTime);
        }
    }
//Add component
    public void AddComponent(Component component)
    {
        components.Add(component);
        component.SetEntity(this);
    }
//Remove Component
    public void RemoveComponent(Component component)
    {
        components.Remove(component);
    }
//Destroy Component
    public void Destroy()
    {
        foreach (var component in components)
        {
            component.OnDestroy();
        }
    }
}
```

```
}
```

The provided implementation of the Entity class is a good starting point for a simple game engine, but it is missing some important features that will become necessary as the complexity of the game grows. For instance, when implementing a system, it is important to be able to retrieve all components of a specific type from an entity. Additionally, the Entity class has an Update method that updates all of its components, which can cause problems with the order of updates. In general, components should be updated in a specific order, such as Input Components, State Components, Transform Components, and then Animated Components. As the game becomes more complex, it will become necessary to implement systems to manage the order of updates and ensure that components are updated in the correct sequence. However, for now, the provided implementation of the Entity class is a good starting point, and updates to individual components can be handled directly in their respective Update methods.

Now, let's take a look at AnimatedComponent:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
namespace MyGame
{
    public class AnimatedComponent : Component
    {
        private string sprite;
        private Rectangle[] frames;
        private int currentFrameIndex;
        private int totalFrames;
        private int frameWidth;
        private int frameHeight;
        private float frameTime;
        private float elapsedFrameTime;
    //Constructor
        public AnimatedComponent(string spriteSheet, int frameWidth, int
frameHeight, int totalFrames, float fps)        {
            this.sprite = spriteSheet;
            this.frameWidth = frameWidth;
            this.frameHeight = frameHeight;
            this.totalFrames = totalFrames;
            this.frameTime = 1/fps;
            this.frames = new Rectangle[totalFrames];
```

```
            this.currentFrameIndex = 0;
        // Calculate the frame rectangles from the sprite sheet
            for (int i = 0; i < totalFrames; i++)
            {
                int x = i * frameWidth;
                frames[i] = new Rectangle(x, 0, frameWidth, frameHeight);
            }
        }
    //Update
        public override void Update(GameTime gameTime)
        {
        // Update the elapsed frame time
            elapsedFrameTime += (float)gameTime.ElapsedGameTime.TotalSeconds;
        // If enough time has passed, move to the next frame
            if (elapsedFrameTime >= frameTime)
            {
                currentFrameIndex++;
                if (currentFrameIndex >= totalFrames)
                {
                    currentFrameIndex = 0;
                }
                elapsedFrameTime = 0;
            }
        }
    //Draw
        public void Draw(SpriteBatch spriteBatch, Vector2 position, int
direction = 1)
        {
            bool isFacingLeft = false;
            if(direction == -1)
                isFacingLeft = true;
            Rectangle currentFrame = frames[currentFrameIndex];
            Texture2D texture = Loader.GetTexture(sprite);
            spriteBatch.Draw(texture,
                    position,
                    currentFrame,
                    Color.White,
                    0f,
                    Vector2.Zero,
                    1f,
                    isFacingLeft ? SpriteEffects.FlipHorizontally :
SpriteEffects.None,
                    0f);
        }
```

```
    //Current Frame
        public Rectangle GetCurrentFrame()
        {
            return frames[currentFrameIndex];
        }
    }
}
```

There are several noteworthy aspects to this class. Firstly, the fps parameter passed to the constructor is used to determine the duration of each frame. It's worth noting that if you obtained your sprite assets from a website like itch.io, the creators may have indicated the intended fps for each animation. If this information is not available, using 20 or 30 fps is generally a safe bet.

Secondly, this class utilizes the Loader class that was implemented earlier to retrieve the sprite sheet. That's why the spritesheet is of type string - the Loader class mapped each texture to a unique string identifier.

Thirdly, this class is designed to handle spritesheets that only have one row. This is intentional since animations can have different sizes and it allows for greater flexibility in design.

Additionally, the Draw method has been implemented to allow for the flipping of spritesheets. By default, the flip is inactive, but this feature is useful because it lets you choose which spritesheets to flip and which ones to load separately. For example, if you have different sprites for left and right animations, but idle and jump only have a few frames, you can flip them without worrying about processing power.

Finally, although this is a component class, it can easily be modified and adapted for object-oriented programming (OOP) systems.

Here is an example of how the AnimatedComponent can be initialized:

```
// Animations (spritesheet name, FRAME width, FRAME height, number of frames,
speed of animation)
            Idle = new AnimatedComponent("player_idle", 32, 32, 11,
GameConstants.FPS);
        // Adding all the components
            AddComponent(Idle);
```

Note that Animated Component doesn’t have a Draw system. You can create a Render system that can

iterate through every object, update their animated components, and then draw the result. I find it useful to render objects based on their state. Here is an example that can hint at the possible implementation:

```
// Draw
        public void Draw(SpriteBatch spriteBatch) {
        // Get the current animation based on the player's current state
        switch(State.currentSuperState){
            case SuperState.OnGround:
                if(State.IsState(ObjectState.WalkLeft))
                    CurrentAnimation = Left;
                else if(State.IsState(ObjectState.WalkRight))
                    CurrentAnimation = Right;
                else if(State.IsState(ObjectState.Idle))
                    CurrentAnimation = Idle;
                break;
            case SuperState.isFalling:
                CurrentAnimation = Fall;
                break;
            case SuperState.isJumping:
                CurrentAnimation = Jump;
                break;
            case SuperState.isDoubleJumping:
                CurrentAnimation = DoubleJump;
                break;
        }
        //Draw the current frame
        if(CurrentAnimation != Left && CurrentAnimation != Right)
            CurrentAnimation.Draw(spriteBatch, Transform.Position,
Transform.direction);
        else
            CurrentAnimation.Draw(spriteBatch, Transform.Position);
        }
```

## Let’s Talk Prototyping

Prototyping is a crucial step in the game development process. It allows developers to quickly test out ideas and mechanics, and determine what works and what doesn't. In this chapter, we will discuss the importance of prototyping and provide some tips for creating effective prototypes.

1. Prototyping is important because it allows developers to test out their ideas and see how they work in practice. By creating a simple prototype, developers can quickly determine if a particular mechanic is fun or engaging, or if it needs to be tweaked or scrapped altogether. This saves time and resources in the long run, as developers can avoid spending hours writing code for mechanics that don't work. Moreover, by creating a tangible prototype, developers can show others how the game will work in a way that is much easier to understand than a verbal or written description.
2. Creating Effective Prototypes

   When creating a digital prototype, it's important to keep things simple. You don't need to create a fully-functional game with all the bells and whistles. Instead, focus on creating a prototype that demonstrates one or two core mechanics. For example, if your game is a platformer, you might create a prototype that shows the player moving and jumping.

   When it comes to graphics, don't worry about creating anything fancy. A simple rectangle or square will suffice. The goal of the prototype is to test out the mechanics, not to create a visually stunning game.

   One key mechanic that you should always include in your prototype is player movement and collisions. This is because movement and collisions are fundamental to almost every game genre, and they can be a bit tricky to get right. By including these mechanics in your prototype, you can quickly determine if they feel smooth and responsive or clunky and frustrating.

Once you have created your prototype, it's important to playtest it thoroughly. Get feedback from others on your team, as well as from players who have never seen the game before. Use this feedback to refine your mechanics and make adjustments as needed.

## Conclusion

In conclusion, I would like to highlight the major tips discussed above:

1. Start small: Begin with a simple game idea that is easy to implement and build upon it gradually. Don't try to create a complex game from the start.
2. Focus on the core gameplay mechanics: Identify the core mechanics that make your game fun and engaging, and focus on developing and refining those mechanics.
3. Break the project into manageable tasks: Divide the development process into smaller, manageable tasks, and focus on completing one task at a time. This will help you stay organized and prevent the project from becoming too complex.
4. Use design documents: Create a design document that outlines the game mechanics, features, and goals. This will help you stay on track and provide a clear roadmap for development.
5. Test often: Test your game frequently and identify areas that need improvement. This will help you identify and fix issues early on in the development process, reducing complexity and preventing major problems down the line.
6. Seek feedback: Get feedback from other students and TAs. This can help identify areas that need improvement and provide valuable insights into the development process.
7. Have fun: Remember to enjoy the process of creating your game! Aim to have as much fun creating it as you do playing it.

I hope these tips will help you successfully create your game!