CE318: Games Console Programming

Lecture 2: A Complete 2D Game

Philipp Rohlfshagen

prohlf@essex.ac.uk
Office 2.529

Outline

1. Code Organisation
2. User Input
3. A 2D Game: Shooter
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

Outline

1. Code Organisation
2. User Input
3. A 2D Game: SHOOTER
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

Programming the OO-Way

In the SquareChase game, all code was placed in the `Game1` class. This is fine for a small game but leads to many problems for anything larger:

- Code becomes unreadable
- Game becomes difficult to debug
- Impossible to reuse code
- Collaborations between developers becomes complicated

It is thus important to adopt a proper object-oriented design where individual segments of code are separated and reused as efficiently as possible. C# is, of course, an object-oriented programming language and thus highly suitable for this approach.

Typical examples for objects in games include *Player*, *Enemy*, *Camera*, *Obstacle*, *Bullet* etc. We will discuss these shortly when it is time to develop a complete game.

XNA Project Templates

Code can be organised in numerous ways in VS2010:

- It is possible to group assets and classes into folders
- Assets may be added as link (useful if they tend to change over time)
- It is also possible to have multiple code projects in a single solution
- One can utilise code from one project in another with `using`
- The project of interest needs to be set as *StartUp Project*
- Windows Games may be converted to Xbox games
  - Files are shared (changes to one project affect the other)
  - Do not forget to set the StartUp project

The **Windows Game Library** template is specifically for code common to multiple projects and allows efficient reuse of code.

Within each project, the use of **Game Components** and **Game Services** can be very useful.

Game Components

Game components provide a modular way of adding functionality to a game:

- Right-click on code project: *Add* → *New Item...* → `GameComponent`

There are two types of game components:

- `GameComponent` – standard game component
- `DrawableGameComponent` – if component is to be shown on screen

Utilise the game component:

- Game logic is added by overriding `Initialize()`, `Update()` and `Draw()`
- A game component is registered using `Game.Components.Add()`
- Methods `Update()` and `Draw()` are called once every frame

It is possible to obtain similar behaviour by calling an object's `Update()` and `Draw()` methods explicitly from `Game1.Update()` and `Game1.Draw()`.

Game Component Template

XNA provides the following template for game components:

```
namespace GameComps
{
  public class MyGameComponent : Microsoft.Xna.Framework.GameComponent
  {
    public MyGameComponent(Game game) : base(game)
    {
      // TODO: Construct any child components here
    }

    public override void Initialize()
    {
      // TODO: Add your initialization code here
      base.Initialize();
    }

    public override void Update(GameTime gameTime)
    {
      // TODO: Add your update code here
      base.Update(gameTime);
    }
  }
}
```

Can change `Microsoft.Xna.Framework.GameComponent` to `Microsoft.Xna.Framework.DrawableGameComponent` and add `Draw()` method.

Game Services

Game services are a mechanism for maintaining loose coupling between objects that need to interact with each other:

- Service providers *register* with `Game.Services`: `Game.Services.AddService`
- Service consumers *request* from `Game.Services`: `Game.Services.GetService`

This arrangement allows objects that require a service to request the service without knowing the name of the service provider.

To allow game components access to `SpriteBatch`, register it in `Game1`:

```
Services.AddService( typeof( SpriteBatch ), spriteBatch );
```

To retrieve `SpriteBatch` in game component:

```
SpriteBatch spriteBatch = (SpriteBatch)Game.Services.GetService(typeof
     (SpriteBatch));
```

Finally, for more fine-grained control, you can set the update order and draw order of all your components. You can also set the `Visible` property to prevent drawing.

Outline

1. Code Organisation
2. User Input
3. A 2D Game: SHOOTER
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

User Input: PC vs Phone vs Console

The way the user interacts with the game is one of the biggest differences between the supported platforms:

- Xbox: gamepad, game-specific input
- PC: mouse+keyboard, gamepad, game-specific input
- Phone: touchscreen, accelerometer

It is possible to account for this in different ways:

- XNA ignores input methods that do not apply
- Possible to use directives to distinguish between platforms
- Important to design game with available inputs in mind

User Input: Nature of Input

Digital:

- Keys of keyboard
- Buttons of gamepad
- DPad of gamepad

Analog:

- Mouse (but not the buttons)
- ThumbSticks of gamepad
- Triggers

Even-based inputs cause a response by the game / program whenever an input event is provided. Due to the nature of the XNA game loop, XNA utilises a polling method to gather a user's inputs.

Input devices are usually polled at the start of `Update()`. It is usually useful to save the state of the input device as well as the time of the last change.

XNA provides, in addition to support for mouse, keyboard and gamepad, a `GamePadType` enumeration that contains: `Guitar`, `DrumKit`, `DancePad`, `Wheel` and `FlightStick` amongst others.

Keyboard

It is possible to use keyboards / chatpads with the Xbox via USB.

To capture keyboard input we use the `Keyboard` class, using `Keyboard.GetState()`:

```
KeyboardState curKbState = Keyboard.GetState();
```

To check if key *A* has been pressed, we use:

```
if ( curKeyboardState.IsKeyDown(Keys.A) )
```

As the game updates 60 times a second, even a brief depression of *A* may cause multiple events to be triggered. We can use the last keyboard state to check for new events:

```
KeyboardState lastKbState;
```

In `Update()`:

```
lastKeyboardState = currentKeyboardState;
```

Then we can check single events as follows:

```
if (curKbState.IsKeyDown(Keys.A) && lastKbState.IsKeyUp(Keys.A))
```

Mouse

Xbox does not support mice so they are only available on PCs.

To capture mouse input we use the Mouse class, using Mouse.GetState():

```
MouseState curMState = Mouse.GetState();
```

The MouseState has the following properties:

X	int	The horizontal position of the mouse
Y	int	The vertical position of the mouse
ScrollWheelValue	int	The scroll position of the mouse wheel
LeftButton	ButtonState	State of the left mouse button
RightButton	ButtonState	State of the right mouse button
MiddleButton	ButtonState	State of the middle mouse button
XButton1	ButtonState	State of extra mouse button 1
XButton2	ButtonState	State of extra mouse button 2

Again, saving the previous state allows one to determine actions by the user:

```
if (curMState.LeftButton == ButtonState.Pressed && lastMState.
     LeftButton == ButtonState.Released)
```

Game Pad

The gamepad is the most important type of input for the Xbox and has both digital and analog controls. Gamepads also allow for force feed back using two vibration motors.

To capture gamepad input we use the `GamePad` class, using `GamePad.GetState()`:

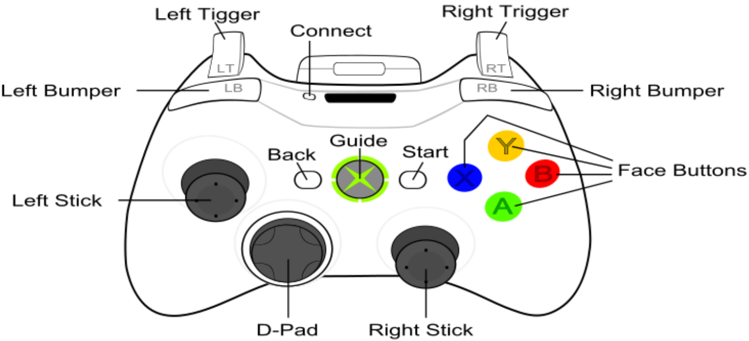
```
GamePadState curGPState = GamePad.GetState(PlayerIndex.One);
```

Here we supply the index of the player whose input we are interested in (this index can also be used for keyboards). The `GamePadState` has the following properties:

`Buttons`	`GamePadButtons`	State of all gamepad buttons
`DPad`	`GamePadDPad`	State of all DPad buttons
`ThumbSticks`	`GamePadThumbSticks`	Position values for left and right thumb sticks
`Triggers`	`GamePadTriggers`	Position of the left and right triggers
`IsConnected`	`bool`	Returns true if the gamepad is connected
`PacketNumber`	`int`	Identifier

From these, all additional attributes may be queried (e.g., position, depression).

Game Pad


Left Tigger
Connect
Right Trigger
LT
RT
Left Bumper
LB
RB
Right Bumper
Back
Guide
Start
Y
X
B
A
Face Buttons
Left Stick
D-Pad
Right Stick

Outline

1. Code Organisation

2. User Input

3. A 2D Game: SHOOTER
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics

4. Summary

A Complete 2D Game

We will now implement a “proper” 2D game in XNA. This tutorial follows very closely the game development tutorial provided by Microsoft on the App Hub:

`http://create.msdn.com/en-US/education/tutorial/2dgame/getting_started`

The game is a simple shooting game where a player needs to navigate a field of enemies, destroying them for points.

You will be asked to implement and extend this game in the labs: you are encouraged to write all the code yourself, following the instructions and not to just download the full version of the game. This is essential to fully understand the details of the implementation and to extend the code later on.

Outline

1. Code Organisation
2. User Input
3. A 2D Game: Shooter
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

Game Design

Game development is a big challenge as even in the drafting stages, games are complex. When you design a game, there is a set of questions you should ask yourself first. These may include:

- What kind of game is it?
- What is the game objective?
- What are the gameplay elements?
- What art assets do you need?
- etc.

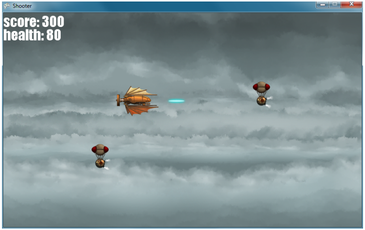
Good game design is essential:

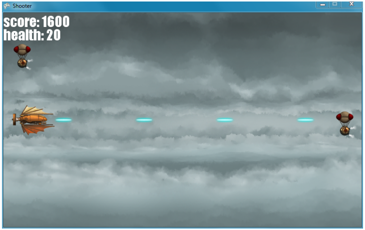
- Helps alleviate potential pitfalls when developing your game
- Standardises and communicated design questions/answers
- Amount of content in the design varies greatly from game to game

Generally, good game design ensure smoother game development.

Game Design

The game is a 2D shooter where the user navigates a ship to shoot enemies approaching from the right; shooting occurs all the time.

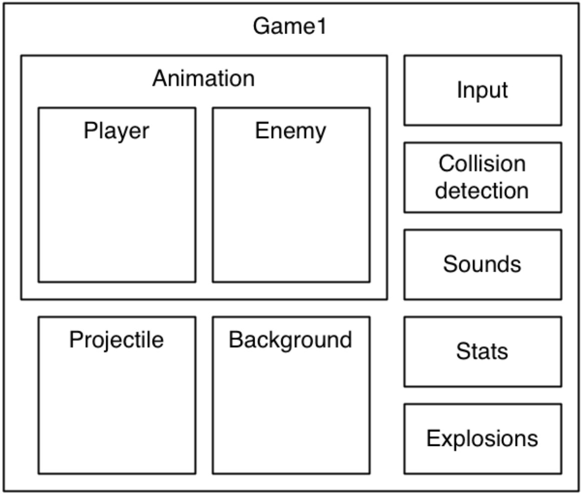




- What kind of game is it?
- Who is the target audience?
- What is the game objective?
- What are the controls?

- What are the gameplay elements?
- What components would we need?
- What code can we potentially re-use?
- What art assets do we need?

Components of Shooter



Getting Started

Assume a new Windows Game project with a namespace **Shooter**. Due to the lack of space, the original code from the tutorial has been somewhat shortened. Class definitions and imports have been omitted.

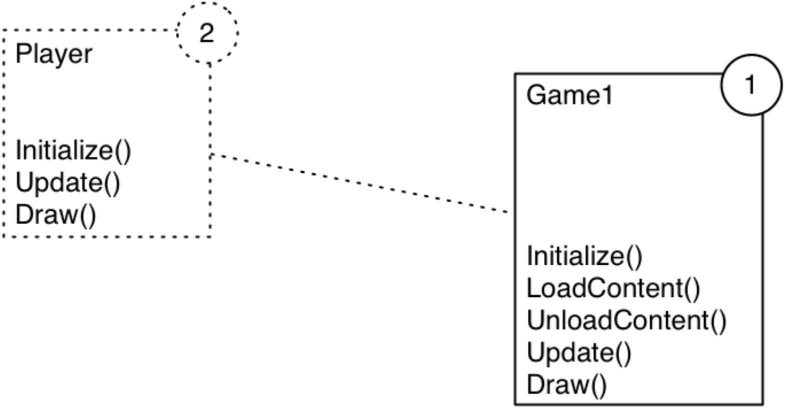
The assets required for this project are as follows:

- player.png - image of the player
- shipAnimation.png - sprite sheet for animated player
- mine.png - image for opponent
- mineAnimation.png - sprite sheet for animated opponent
- explosion.png - sprite sheet for animated explosion
- laser.png - image for shooting projectile
- mainBackground.png - the main background image
- bgl1.png & bgl2.png - the layers for the parallaxing background
- gameMusic.mp3 - background music
- explosion.wav - sound of explosion
- laserFire.wav - sound of shooting

Outline

1. Code Organisation
2. User Input
3. A 2D Game: Shooter
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

The Game So Far



Creating a Player (1/3)

First we create the player in a class called `Player` and add the following:

```
public Texture2D PlayerTexture;
public Vector2 Position;
public bool Active
public int Health;

public void Initialize() { }
public void Update() { }
public void Draw() { }

public int Width
{
  get { return PlayerTexture.Width; }
}

public int Height
{
  get { return PlayerTexture.Height; }
}
```

This provides the basic structure for our player, with some attributes. The use of `Initalize()`, `Update()` and `Draw()` is very common for the different game objects. Could use game components also.

Creating a Player (2/3)

We now need to fill in the methods of `Player`.

```
public void Initialize(Texture2D texture, Vector2 position)
{
  PlayerTexture = texture;
  Position = position;
  Active = true;
  Health = 100;
}
```

This initialises the player object and allows it to be drawn onto the screen. Note how the attribute for the texture and position are passed into the `Initialize()` method – these come from `Game1`.

```
public void Draw(SpriteBatch spriteBatch)
{
  spriteBatch.Draw(PlayerTexture, Position, null, Color.White, 0f,
       Vector2.Zero, 1f, SpriteEffects.None, 0f);
}
```

Source rectangle is not required, hence `null`. The full set of arguments is: texture, position, source rectangle, tinting colour, rotation, rotation origin, scale, effect and layer.

Creating a Player (3/3)

Now we need to add the player to the game and control its state from there.

In `Game1`, we add

```
Player player;
```

In `Game1.Initialize()`:

```
player = new Player();
```

In `Game1.LoadContent()`:

```
Vector2 playerPosition = new Vector2(GraphicsDevice.Viewport.
     TitleSafeArea.X, GraphicsDevice.Viewport.TitleSafeArea.Y +
     GraphicsDevice.Viewport.TitleSafeArea.Height / 2);

player.Initialize(Content.Load<Texture2D>("player"), playerPosition);
```

Q What is the TitleSafeArea?

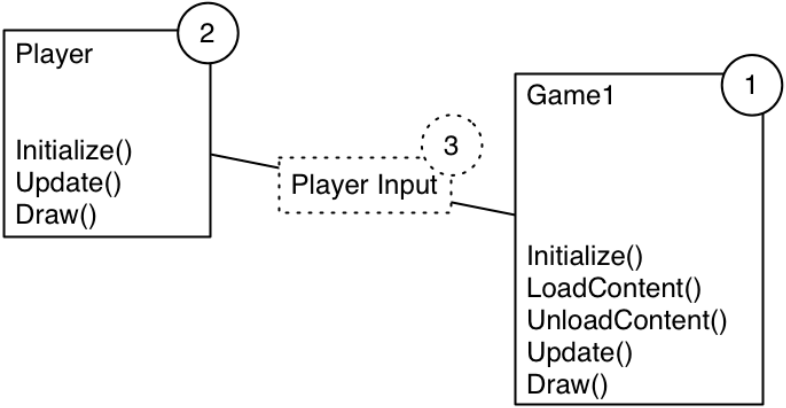
In `Game1.Draw()`:

```
spriteBatch.Begin();
player.Draw(spriteBatch);
spriteBatch.End();
```

Outline

1. Code Organisation
2. User Input
3. A 2D Game: SHOOTER
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

The Game So Far



Processing Player Inputs (1/3)

The player object is controlled by the person playing the game. We thus need to get some inputs.

In `Game1`, we add:

```
KeyboardState currentKeyboardState;
GamePadState currentGamePadState;

float playerMoveSpeed = 8.0f;
```

Q Why don't we save the previous input state?

To update the player object, we create a new method called `UpdatePlayer()`. We first deal with thumb-stick movements:

```
player.Position.X += currentGamePadState.ThumbSticks.Left.X *
     playerMoveSpeed;
player.Position.Y -= currentGamePadState.ThumbSticks.Left.Y *
     playerMoveSpeed;
```

Processing Player Inputs (2/3)

Then we deal with keyboard/D-pad movements:

```
if (currentKeyboardState.IsKeyDown(Keys.Left) || currentGamePadState.DPad.Left == ButtonState.Pressed)
{
  player.Position.X -= playerMoveSpeed;
}

if (currentKeyboardState.IsKeyDown(Keys.Right) || currentGamePadState.DPad.Right == ButtonState.Pressed)
{
  player.Position.X += playerMoveSpeed;
}

if (currentKeyboardState.IsKeyDown(Keys.Up) || currentGamePadState.DPad.Up == ButtonState.Pressed)
{
  player.Position.Y -= playerMoveSpeed;
}

if (currentKeyboardState.IsKeyDown(Keys.Down) || currentGamePadState.DPad.Down == ButtonState.Pressed)
{
  player.Position.Y += playerMoveSpeed;
}
```

Can press key continuously for continuous motion. No previous state required.

Processing Player Inputs (3/3)

After adjusting the player's position, we need to make sure it is within bounds:

```
player.Position.X = MathHelper.Clamp(player.Position.X, 0,
     GraphicsDevice.Viewport.Width);
player.Position.Y = MathHelper.Clamp(player.Position.Y, 0,
     GraphicsDevice.Viewport.Height);
```

The `MathHelper` class is a utility class that contains numerous useful functions (especially for 3D graphics as will be shown in the next lecture).

We then utilise the new method in `Game1.Update()`:

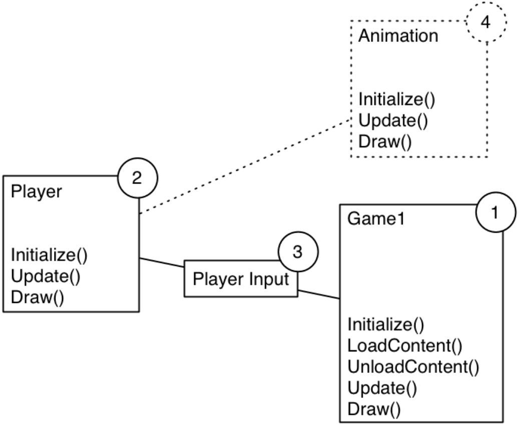
```
currentKeyboardState = Keyboard.GetState();
currentGamePadState = GamePad.GetState(PlayerIndex.One);

UpdatePlayer(gameTime);
```

We now have full control over the player, using the keyboards direction keys, the thump pads or the DPad.

Outline

1. Code Organisation
2. User Input
3. A 2D Game: SHOOTER
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

The Game So Far



Animating Players (1/5)

We now make use of sprite sheets to animate the player. We create a new class called `Animation`. This is a more general class that we can also use to animate other objects based on sprite sheets.

Add the following fields to the class:

```
Texture2D spriteStrip;
float scale;
int elapsedTime;
int frameTime;
int frameCount;
int currentFrame;
Color color;
Rectangle sourceRect = new Rectangle();
Rectangle destinationRect = new Rectangle();
public int FrameWidth;
public int FrameHeight;
public bool Active;
public bool Looping;
public Vector2 Position;

public void Initialize() { }
public void Update() { }
public void Draw() { }
```

Note: code relies on fact that sprite sheet is a strip (i.e., 1 × n).

Animating Players (2/5)

Again, just like before, we now need to fill in the empty methods.

For `Animation.Initialize()`:

```
public void Initialize(Texture2D texture, Vector2 position, int frameWidth, int frameHeight, int frameCount, int frametime, Color color, float scale, bool looping)
{
  this.color = color;
  this.FrameWidth = frameWidth;
  this.FrameHeight = frameHeight;
  this.frameCount = frameCount;
  this.frameTime = frametime;
  this.scale = scale;

  Looping = looping;
  Position = position;
  spriteStrip = texture;

  elapsedTime = 0;
  currentFrame = 0;

  Active = true;
}
```

This sets all the variables required for the animated player object.

Animating Players (3/5)

Now we create the code for `Animation.Update()`:

```
public void Update(GameTime gameTime)
{
  if (Active == false)
    return;

  elapsedTime += (int)gameTime.ElapsedGameTime.TotalMilliseconds;

  if (elapsedTime > frameTime)
  {
    if (currentFrame++ == frameCount)
    {
      currentFrame = 0;

      if (Looping == false)
        Active = false;
    }

    elapsedTime = 0;
  }

  sourceRect = new Rectangle(currentFrame * FrameWidth, 0, FrameWidth,
        FrameHeight);

  destinationRect = new Rectangle((int)Position.X - (int)(FrameWidth *
        scale) / 2, (int)Position.Y - (int)(FrameHeight * scale) / 2,
       (int)(FrameWidth * scale), (int)(FrameHeight * scale));
}
```

Animating Players (4/5)

Finally, we fill in `Animation.Draw()`:

```
public void Draw(SpriteBatch spriteBatch)
{
  if (Active)
  {
    spriteBatch.Draw(spriteStrip, destinationRect, sourceRect, color);
  }
}
```

In order to make use of the animation, we replace `PlayerTexture` in `Player` with

```
public Animation PlayerAnimation;
```

and fix all dependencies in the code.

We also need to write an `Update()` method for `Player`:

```
public void Update(GameTime gameTime)
{
  PlayerAnimation.Position = Position;
  PlayerAnimation.Update(gameTime);
}
```

Note: the player's position is updated in `Game1` and the animation's position is updated in `Player.Update()`.

Animating Players (5/5)

Finally, we update the call to `player.Initialize()` in `Game1` with

```
Animation playerAnimation = new Animation();

Texture2D playerTexture = Content.Load<Texture2D>("shipAnimation");

Vector2 playerPosition = new Vector2 (GraphicsDevice.Viewport.
     TitleSafeArea.X, GraphicsDevice.Viewport.TitleSafeArea.Y +
     GraphicsDevice.Viewport.TitleSafeArea.Height / 2);

playerAnimation.Initialize(playerTexture, Vector2.Zero, 115, 69, 8,
     30, Color.White, 1f, true);

player.Initialize(playerAnimation, playerPosition);
```

In `UpdatePlayer()` we need to add at the very beginning the line:

```
player.Update(gameTime);
```

Note: `Game1.Update()` calls `player.Update()` which calls `PlayerAnimation.Update()`.

We now have an animated player and can use the class `Animation` for other animations also.

Outline

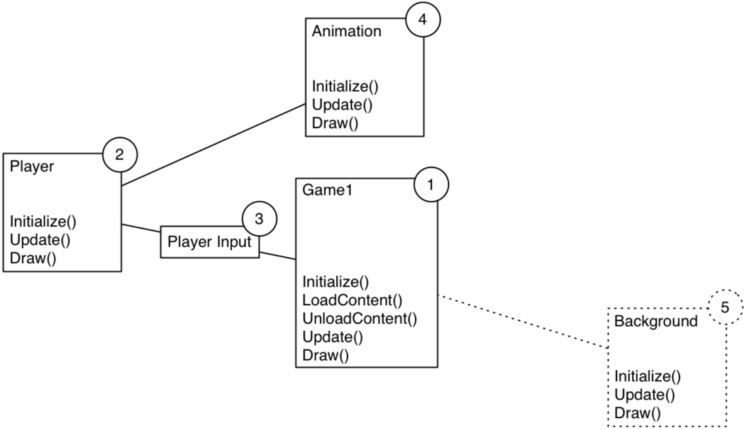
1. Code Organisation

2. User Input

3. A 2D Game: SHOOTER
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics

4. Summary

The Game So Far



Drawing a Background (1/4)

The background for Shooter illustrates yet another type of animation that may be used with 2D games. The background is drawn using 3 layers with transparent sections. Moving the top two layers at different speeds creates the illusion of depth. This is known as a **parallax background**.

In order to do this, we first create a new class called `ParallaxingBackground`:

```
Texture2D texture;
Vector2 [] positions;
int speed;

public void Initialize() { }
public void Update() { }
public void Draw() { }
```

Once again, this component of the game makes use of the typical initialise, update and draw methods. This class will be used in `Game1` to create two moving objects on top of a static background image.

Drawing a Background (2/4)

In `ParallaxingBackground.Initialize()`, we put the following code:

```
public void Initialize(ContentManager content, String texturePath, int screenWidth, int speed)
{
  texture = content.Load<Texture2D>(texturePath);
  this.speed = speed;
  positions = new Vector2[screenWidth / texture.Width + 1];

  for (int i = 0; i < positions.Length; i++)
  {
    positions[i] = new Vector2(i * texture.Width, 0);
  }
}
```

This initialises an array of position for use with **tiling**. Tiling is used commonly in games. **Q** Why?

The `ParallaxingBackground.Draw()` method is defined as follows:

```
public void Draw(SpriteBatch spriteBatch)
{
  for (int i = 0; i < positions.Length; i++)
  {
    spriteBatch.Draw(texture, positions[i], Color.White);
  }
}
```

Drawing a Background (3/4)

Finally, we place the following code into `ParallaxingBackground.Update()`:

```
public void Update()
{
  for (int i = 0; i < positions.Length; i++)
  {
    positions[i].X += speed;

    if (speed <= 0)
    {
      if (positions[i].X <= -texture.Width)
      {
        positions[i].X = texture.Width * (positions.Length - 1);
      }
    }
    else
    {
      if (positions[i].X >= texture.Width * (positions.Length - 1))
      {
        positions[i].X = -texture.Width;
      }
    }
  }
}
```

This method first adds the speed and then checks the bounds for both positive and negative speeds. It allows for parts of the sprite to be drawn beyond the frame of the game.

Drawing a Background (4/4)

Inside `Game1`:

```
Texture2D mainBackground;
ParallaxingBackground bgl1;
ParallaxingBackground bgl2;
```

Inside `Game1.Initialize()`:

```
bgl1 = new ParallaxingBackground();
bgl2 = new ParallaxingBackground();
```

Inside `Game1.LoadContent()`:

```
bgl1.Initialize(Content, "bgl1", GraphicsDevice.Viewport.Width, -1);
bgl2.Initialize(Content, "bgl2", GraphicsDevice.Viewport.Width, -2);

mainBackground = Content.Load<Texture2D>("mainbackground");
```

Inside `Game1.Update()`:

```
bgl1.Update();
bgl2.Update();
```

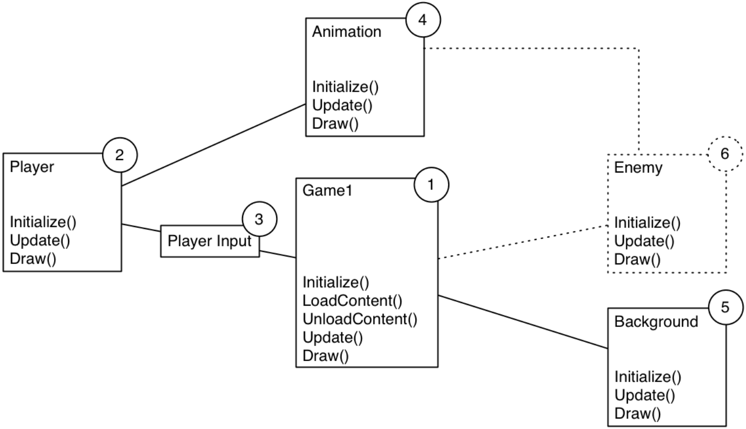
Inside `Game1.Draw()`:

```
spriteBatch.Draw(mainBackground, Vector2.Zero, Color.White);

bgl1.Draw(spriteBatch);
bgl2.Draw(spriteBatch);
```

Outline

1. Code Organisation
2. User Input
3. A 2D Game: SHOOTER
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

The Game So Far



Adding Enemies (1/5)

Add a new class for the enemies, `Enemy`:

```csharp
public Animation EnemyAnimation;
public Vector2 Position;
public bool Active;
public int Health;
public int Damage;
public int Value;

public int Width
{
  get { return EnemyAnimation.FrameWidth; }
}

public int Height
{
  get { return EnemyAnimation.FrameHeight; }
}

float enemyMoveSpeed;

public void Initialize() { }
public void Update() { }
public void Draw() { }
```

An enemy is quite similar to a player except that it is controlled by the game engine and not the gamer. You could add some AI here later.

Adding Enemies (2/5)

Now implement the missing methods:

`Enemy.Initialize()`:

```
public void Initialize(Animation animation,Vector2 position)
{
  EnemyAnimation = animation;
  Position = position;
  Active = true;
  Health = 10;
  Damage = 10;
  enemyMoveSpeed = 6f;
  Value = 100;
}
```

`Enemy.Update()`:

```
public void Update(GameTime gameTime)
{
  Position.X -= enemyMoveSpeed;
  EnemyAnimation.Position = Position;
  EnemyAnimation.Update(gameTime);

  if (Position.X < -Width || Health <= 0)
  {
    Active = false;
  }
}
```

Note: we subtract the `enemyMoveSpeed` as enemies move right to left.

Adding Enemies (3/5)

Finally we implement `Enemy.Draw()`:

```
public void Draw(SpriteBatch spriteBatch)
{
  EnemyAnimation.Draw(spriteBatch);
}
```

This just makes use of the animation's draw method.

Now it is time to add numerous enemies to the game. In `Game1` add:

```
Texture2D enemyTexture;
List<Enemy> enemies;

TimeSpan enemySpawnTime;
TimeSpan previousSpawnTime;

Random random;
```

Here we use a list to store multiple enemies and make use of `TimeSpan` to control the rate at which enemies are added to the list. The random number generator is used to determine the initial vertical position of the enemy.

Adding Enemies (4/5)

In `Game1.Initialize()`:

```
enemies = new List<Enemy> ();

previousSpawnTime = TimeSpan.Zero;
enemySpawnTime = TimeSpan.FromSeconds(1.0f);

random = new Random();
```

In `Game1.LoadContent()`:

```
enemyTexture = Content.Load<Texture2D>("enemy");
```

We then add a new method to `Game1`:

```
private void AddEnemy()
{
  Animation enemyAnimation = new Animation();

  enemyAnimation.Initialize(enemyTexture, Vector2.Zero, 47, 61, 8, 30,
       Color.White, 1f, true);

  Vector2 position = new Vector2(GraphicsDevice.Viewport.Width +
       enemyTexture.Width / 2, random.Next(100, GraphicsDevice.
       Viewport.Height -100));

  Enemy enemy = new Enemy();
  enemy.Initialize(enemyAnimation, position);
  enemies.Add(enemy);
}
```

Adding Enemies (5/5)

We then add another method to `Game1`:

```
private void UpdateEnemies(GameTime gameTime)
{
  if (gameTime.TotalGameTime - previousSpawnTime > enemySpawnTime)
  {
    previousSpawnTime = gameTime.TotalGameTime;
    AddEnemy();
  }

  for (int i = enemies.Count - 1; i >= 0; i--)
  {
    enemies[i].Update(gameTime);

    if (enemies[i].Active == false)
    {
      enemies.RemoveAt(i);
    }
  }
}
```

In `Game1.Update()`:

```
UpdateEnemies(gameTime);
```

In `Game1.Draw()`:

```
for (int i = 0; i < enemies.Count; i++)
{
  enemies[i].Draw(spriteBatch);
}
```

Outline

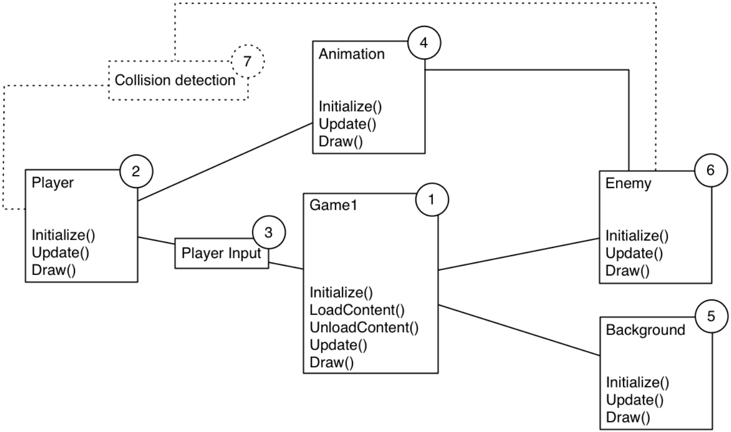
1. Code Organisation

2. User Input

3. A 2D Game: Shooter
    - Game Design
    - Creating a Player
    - Processing Player Inputs
    - Animating Players
    - Drawing the Background
    - Adding Enemies
    - Simple Collision Detection
    - Shooting
    - Explosions
    - Playing Sounds
    - Adding Game Statistics

4. Summary

The Game So Far



Checking for Collisions (1/2)

To check for collisions, we do not need another class. Instead, we place a new method inside `Game1`:

```
private void UpdateCollision()
{
  Rectangle rectangle1;
  Rectangle rectangle2;

  rectangle1 = new Rectangle((int)player.Position.X, (int)player.Position.Y, player.Width, player.Height);

  for (int i = 0; i <enemies.Count; i++)
  {
    rectangle2 = new Rectangle((int)enemies[i].Position.X, (int)enemies[i].Position.Y, enemies[i].Width, enemies[i].Height);

    if(rectangle1.Intersects(rectangle2))
    {
      player.Health -= enemies[i].Damage;
      enemies[i].Health = 0;

      if (player.Health <= 0)
        player.Active = false;
    }
  }
}
```

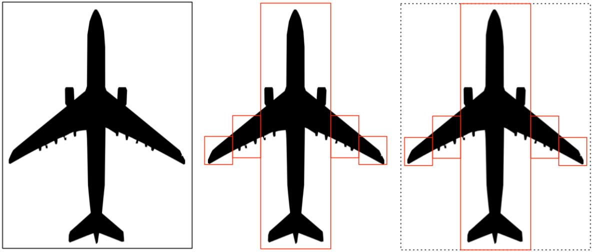
We create a **bounding box** for the player and each enemy, then check if they overlap.

Checking for Collisions (2/2)

Add to `Game1.Update()`:

```
UpdateCollision();
```

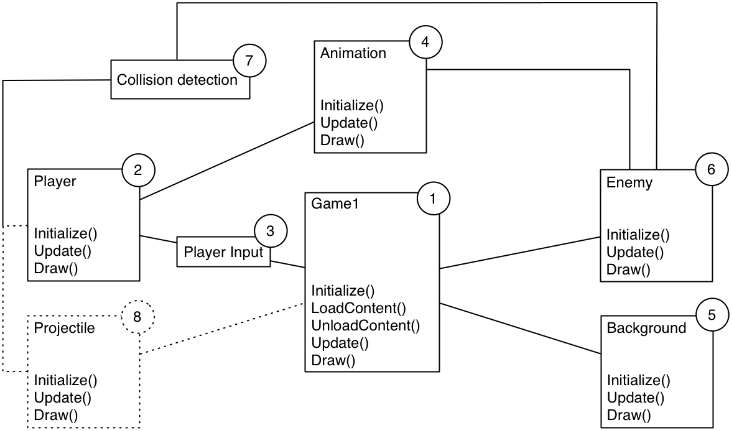
We now have complete collision detection for our game. We have constructed simple bounding rectangles and then simply check for all of them whether they overlap (`Intersects()`). Collision detection will be a recurring theme throughout this course and will be covered in much greater depth in lecture 7 and also in lecture 10 (when we discuss performance).



Outline

1. Code Organisation
2. User Input
3. A 2D Game: Shooter
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

The Game So Far



Creating Projectiles (1/5)

We create a new class called `Projectile` to represent bullets:

```
public Texture2D Texture;
public Vector2 Position;
public bool Active;
public int Damage;
Viewport viewport;

public int Width
{
  get { return Texture.Width; }
}

public int Height
{
  get { return Texture.Height; }
}

float projectileMoveSpeed;

public void Initialize() { }
public void Update() { }
public void Draw() { }
```

`Projectile` is the third kind of moving entity in our game, besides `Player` and `Enemy`; the class structure is thus very similar.

The `Viewport` represents the viewable boundary of the game.

Creating Projectiles (2/5)

Fill in `Projectile.Initialize()`:

```
public void Initialize(Viewport viewport, Texture2D texture, Vector2 position)
{
  Texture = texture;
  Position = position;
  this.viewport = viewport;
  Active = true;
  Damage = 2;
  projectileMoveSpeed = 20f;
}
```

Fill in `Projectile.Update()`:

```
public void Update()
{
  Position.X += projectileMoveSpeed;

  if (Position.X + Texture.Width / 2 > viewport.Width)
    Active = false;
}
```

Fill in `Projectile.Draw()`:

```
public void Draw(SpriteBatch spriteBatch)
{
  spriteBatch.Draw(Texture, Position, null, Color.White, 0f, new Vector2(Width / 2, Height / 2), 1f, SpriteEffects.None, 0f);
}
```

Creating Projectiles (3/5)

In `Game1` we add:

```
Texture2D projectileTexture;
List<Projectile> projectiles;

TimeSpan fireTime;
TimeSpan previousFireTime;
```

In `Game1.Initialize()` we add:

```
projectiles = new List<Projectile>();
fireTime = TimeSpan.FromSeconds(0.15f);
```

In `Game1.LoadContent()`:

```
projectileTexture = Content.Load<Texture2D>("laser");
```

Finally, we add a new method `AddProjectile()`:

```
private void AddProjectile(Vector2 position)
{
  Projectile projectile = new Projectile();
  projectile.Initialize(GraphicsDevice.Viewport, projectileTexture,
      position);
  projectiles.Add(projectile);
}
```

Creating Projectiles (4/5)

In `Game1.UpdatePlayer()`:

```
if (gameTime.TotalGameTime - previousFireTime > fireTime)
{
  previousFireTime = gameTime.TotalGameTime;
  AddProjectile(player.Position + new Vector2(player.Width / 2, 0));
}
```

This methods adds new projectiles when it is time.

Similar to before, we create a new method to update the projectiles:

```
private void UpdateProjectiles()
{
  for (int i = projectiles.Count - 1; i >= 0; i--)
  {
    projectiles[i].Update();

    if (projectiles[i].Active == false)
    {
      projectiles.RemoveAt(i);
    }
  }
}
```

which we call from inside `Game1.Update()`:

```
UpdateProjectiles();
```

Creating Projectiles (5/5)

In `Game1.Draw()` we add the following:

```
for (int i = 0; i < projectiles.Count; i++)
{
  projectiles[i].Draw(spriteBatch);
}
```

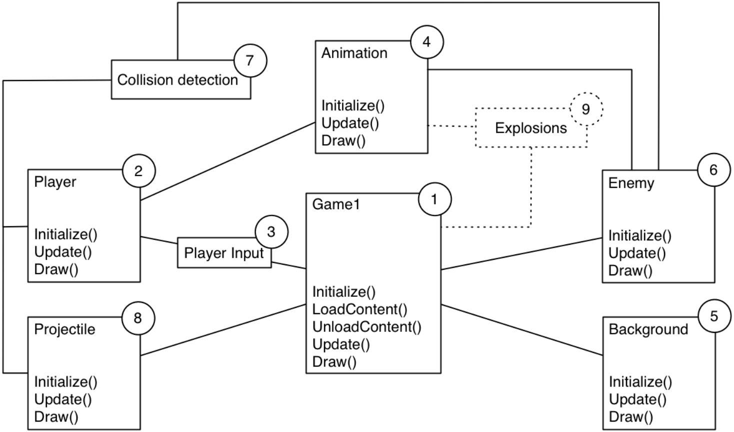
Finally, we add the following to `Game1.UpdateCollision()`:

```
for (int i = 0; i < projectiles.Count; i++)
{
  for (int j = 0; j < enemies.Count; j++)
  {
    rectangle1 = new Rectangle((int)projectiles[i].Position.X -
    projectiles[i].Width / 2,(int)projectiles[i].Position.Y -
    projectiles[i].Height / 2,projectiles[i].Width, projectiles[i].
        Height);

    rectangle2 = new Rectangle((int)enemies[j].Position.X - enemies[j
        ].Width / 2, (int)enemies[j].Position.Y - enemies[j].Height /
         2, enemies[j].Width, enemies[j].Height);

    if (rectangle1.Intersects(rectangle2))
    {
      enemies[j].Health -= projectiles[i].Damage;
      projectiles[i].Active = false;
    }
  }
}
```

Outline

1. Code Organisation
2. User Input
3. A 2D Game: Shooter
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

The Game So Far



Explosions (1/2)

Explosions make use of the class `Animation` and all code is contained in `Game1`. First we add the animations to the game:

```
Texture2D explosionTexture;
List<Animation> explosions;
```

and instantiate that list of animations:

```
explosions = new List<Animation>();
```

In `Game1.LoadContent()`:

```
explosionTexture = Content.Load<Texture2D>("explosion");
```

We then create a new method called `AddExplosion()`:

```
private void AddExplosion(Vector2 position)
{
  Animation explosion = new Animation();
  explosion.Initialize(explosionTexture,position, 134, 134, 12, 45,
       Color.White, 1f, false);
  explosions.Add(explosion);
}
```

Explosions (2/2)

We then need to actually instantiate explosions in `Game1.UpdateEnemies()`:

```
if (enemies[i].Health <= 0)
{
  AddExplosion(enemies[i].Position);
}
```

We then need to update all current explosions:

```
private void UpdateExplosions(GameTime gameTime)
{
  for (int i = explosions.Count - 1; i >= 0; i--)
  {
    explosions[i].Update(gameTime);

    if (explosions[i].Active == false)
    {
      explosions.RemoveAt(i);
    }
  }
}
```

and add to `Game1.Update()`:

```
UpdateExplosions(gameTime);
```

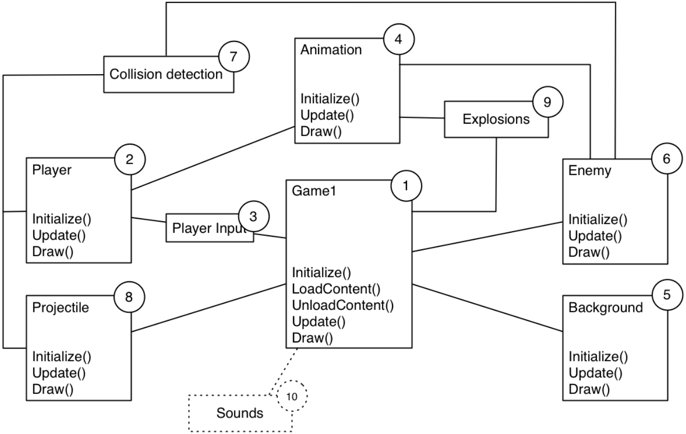
Finally, we extend `Game1.Draw()` with the following:

```
for (int i = 0; i < explosions.Count; i++)
{
  explosions[i].Draw(spriteBatch);
}
```

Outline

1. Code Organisation
2. User Input
3. A 2D Game: Shooter
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

The Game So Far



Playing Sounds (1/1)

We use `SoundEffect` File & Forget and add to `Game1`:

```
SoundEffect laserSound;
SoundEffect explosionSound;
Song gameplayMusic;
```

Load assets in `Game1.LoadContent()` (make sure to check the content importer and processor):

```
gameplayMusic = Content.Load<Song>("sound/gameMusic");
laserSound = Content.Load<SoundEffect>("sound/laserFire");
explosionSound = Content.Load<SoundEffect>("sound/explosion");

PlayMusic(gameplayMusic);
```

Create a new method to play the songs:

```
private void PlayMusic(Song song)
{
  try
  {
    MediaPlayer.Play(song);
    MediaPlayer.IsRepeating = true;
  } catch { }
}
```

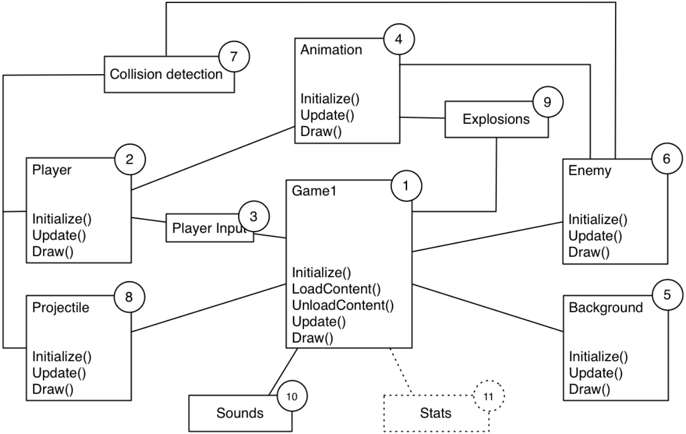
Play sounds when appropriate (in `Game1.UpdatePlayer()` and `Game1.UpdateEnemies`):

```
laserSound.Play();
explosionSound.Play();
```

Outline

1. Code Organisation
2. User Input
3. A 2D Game: Shooter
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

The Game So Far



Adding Stats (1/1)

The player needs to know about his/her health and score.
We need to create a new font and then add the following to `Game1`:

```
int score;
SpriteFont font;
```

In `Game1.Initialize()`:

```
score = 0;
```

and in `Game1.LoadContent()`:

```
font = Content.Load<SpriteFont>("gameFont");
```

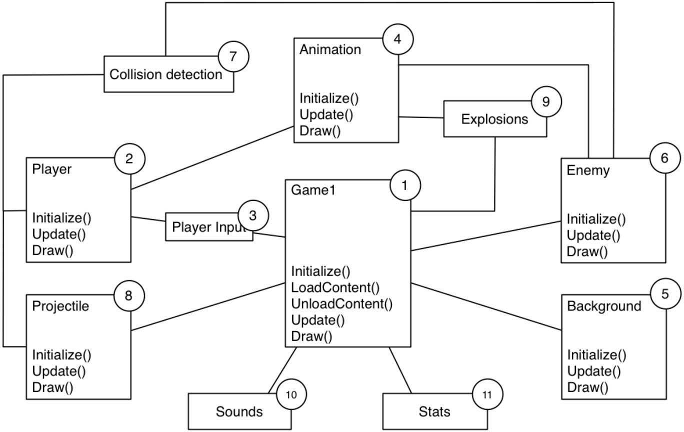
We update the score in `Game1.Update()` every time an enemy dies:

```
score += enemies[i].Value;
```

Finally, we draw in the information on the screen:

```
spriteBatch.DrawString(font, "score: " + score, new Vector2(
    GraphicsDevice.Viewport.TitleSafeArea.X, GraphicsDevice.Viewport.
    TitleSafeArea.Y), Color.White);
spriteBatch.DrawString(font, "health: " + player.Health, new Vector2(
    GraphicsDevice.Viewport.TitleSafeArea.X, GraphicsDevice.Viewport.
    TitleSafeArea.Y + 30), Color.White);
```

The Final Game



Outline

1. Code Organisation
2. User Input
3. A 2D Game: Shooter
   - Game Design
   - Creating a Player
   - Processing Player Inputs
   - Animating Players
   - Drawing the Background
   - Adding Enemies
   - Simple Collision Detection
   - Shooting
   - Explosions
   - Playing Sounds
   - Adding Game Statistics
4. Summary

Summary

In this lecture we looked at a proper way to implement games in XNA, making use of classes and code re-use. We pulled together the different concepts explored so far (sprites, user inputs, 2D animations, sounds, etc.) and created a full 2D game.

We covered

- Code organisation
- Game components
- Game services
- User input
  - gamepad
  - mouse
  - keyboard

- Implemented SHOOTER
  - Game design
  - Content generation
  - 2D animations
  - Text and fonts
  - Sounds
  - User inputs

Lab Preview (1/2)

In the lab you will implement the SHOOTER game from scratch. Once finished, you will extend and improve the game to your liking. We will provide a number of ideas to get you started. These include:

- Add a splash screen and add the ability to pause the game
- Only shoot when pressing a button (e.g. space bar)
- Limit the number of bullets the player has (and add refill packs)
- Make the opponent shoot also
- Add different levels of difficulty
- Add different types of enemies
- Add obstacles

Next lecture: moving from 2D to 3D. We will cover the graphics pipeline, 3D primitives and some basic transformations.

Lab Preview (2/2)

In lab 2, we will set up the Xboxes for deployment of all the code we will write from hereon. Prior to the lab, you are asked to do the following:

- Get a free email account at Microsoft Live
- Register at `www.dreamspark.com`
- Verify your student status at DreamSpark (needs to be done on campus)
- Join the App Hub using the verified email address

A full set of detailed instructions may be found in the slides for lecture 1 and on the course web page. Make sure to complete these steps so we can set up the Xboxes first thing in the labs.