CE318: Games Console Programming

Lecture 10: Multi-Player Games

Philipp Rohlfshagen

prohlf@essex.ac.uk
Office 2.529

Multi-Player Games

Multi-player games have become increasingly popular in recent years. One of the reasons for this is the lack of convincing NPC AI. However, it is also often more rewarding to play against someone you know.

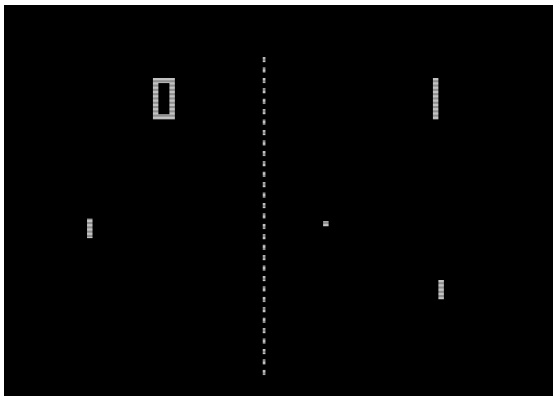
We have multiple types of multi-player games:

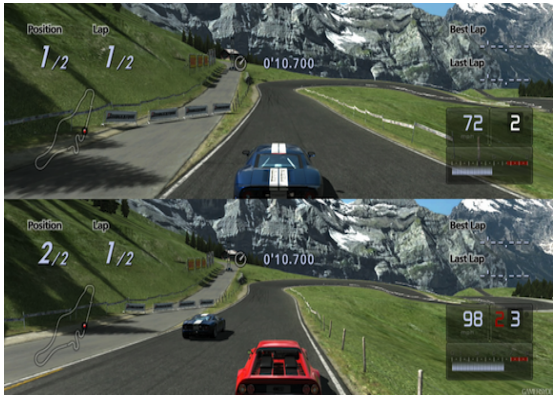
- Shared single screen (usually 2 players)
- Split-screen (usually 2-4 players)
- Networked (LAN or WAN)

The former was very popular prior to the widespread availability of reliable network infrastructures and the rise of massive online communities. Nowadays, some blockbuster games are played exclusively online (MMOs).

Joint-screen or split-screen multi-player games have recently become more popular again due to the rise of novel input controls: Nintendo Wii, PlayStation Move and Xbox Kinect.

Multi-Player Games





Outline

Outline

Split-Screen Considerations

Split-screen games allow 2-4 players to play against one another

- You only use a single screen, console, copy of game
- Multiple input methods (e.g., gamepads) are required

Each player has his/her own view of the game state

- Use independent views / cameras for all players involved
- Consider the game's suitability for split-screen (e.g., racing vs FPS)

One of the biggest issues with split-screen is screen real-estate

- Side by side or top and bottom
- Minimising auxiliary information such as health, track preview etc.

To extend your single-player game to multi-player split-screen, you need to:

- Implement multiple cameras with different instances of `Viewport`
- Keep track of all player objects and their attributes
- Keep track of inputs to ensure they modify the correct objects in the game

Creating a Split-Screen

The `ViewPort` tells the graphics device where to draw the game's graphics:

- The `Viewport` is the 2D rectangle where the 3D scene will be drawn
- By default, the view port is set to the size of the client window

We can create a split-screen game using:

- Multiple instances of `Viewport`, each with own size and location
- Multiple cameras (views **and** projections) for all player's involved

We add a `Viewport` property to our camera class:

```
1 public Viewport viewport { get; set; }
```

We then pass a `Viewport` instance to each camera instance:

```
1 Viewport vp1 = GraphicsDevice.Viewport;
2 Viewport vp2 = GraphicsDevice.Viewport;
3
4 vp1.Height = vp2.Height = (GraphicsDevice.Viewport.Height / 2);
5 vp2.Y = vp1.Height;
```

Q How are the cameras configured? What would the alternative look like?

Creating a Split-Screen

We use two camera objects:

```
1 public Camera camera1, camera2;
```

Each camera has its own `Viewport`.

When we construct the viewing frustum, we need to take the `Viewport` into account. We define the aspect ratio as

```
1 (float)viewport.Width / (float)viewport.Height
```

instead of

```
1 GraphicsDevice.Viewport.AspectRatio
```

Finally, we need to change the `Draw()` method. For each player:

- Use the `Viewport`, view and projection of the current camera

```
1 GraphicsDevice.Viewport = camera1.viewport;
2 playerOneObject.Draw(camera1.view, camera1.proj);
3
4 GraphicsDevice.Viewport = camera2.viewport;
5 playerTwoObject.Draw(camera2.view, camera2.proj);
```

Note: the `Viewport` is set globally and hence affects all methods that draw elsewhere in the game.

Dealing with User Input & Demo

This is all that is required for a split-screen view. The last bit that requires change is the input method.

On the keyboard, we can simply define different sets of keys for each player.

Q Remember how to distinguish multiple gamepads?

Let's have a look at a split-screen implementation. To highlight the different camera perspectives, we can make use of the method

```
1 DebugShapeRenderer.AddBoundingFrustum(BoundingFrustum f, Color c)
```

The bounding frustum is simply:

```
1 BoundingFrustum bf = new BoundingFrustum(view * projection);
```

Final note: you will need to keep track of each player. Hence you will require multiple scores, etc. It is best to approach this using proper OO design, reusing most code. Also make sure to display the correct information for each player.

Q Can you identify a scenario where split-screen would ruin the experience?

Outline

Networked Multi-Player Games

The first step in creating a network game is the type of network to be used:

- Peer-to-peer
  - All participants are clients of one another
  - Each participant sends updates to all other participants
- Client/server
  - One (or more) servers
  - All participants are clients
  - Server receives actions from all participants and updates game state
  - Server then sends updates to all participants

It is also possible to use a hybrid of the two. The client/server model requires fewer transmissions overall but is prone to single point failure (i.e., if the server goes down, no one can play). Also, the server needs to be able to deal with all connections simultaneously.

The choice of network depends on the type of game (e.g., speed of updates) and the number of players.

Networked Multi-Player Games

To network games, we make use of

```
1 Microsoft.Xna.Framework.Net
```

We also need to add the networking and gamer services to the game:

```
1 Components.Add(new GamerServicesComponent(this));
```

The XNA networking API makes use of **Games for Windows LIVE** and **Xbox LIVE**. The former makes use of the same gamertags and online identities as the latter.

Wee need to run at least 2 instances of the code for testing:

- Run the code as a Windows Game on the PC
- Execute the same code as an Xbox Game on the Xbox 360

Note: the LIVE network requires both gamertags to have App Hub memberships. To overcome this issue, the networking can be done over LAN using `SystemLink`. Then only the Xbox gamer requires an App Hub membership (which is required to deploy code on the Xbox anyway). The Windows gamer should use a **local profile**; see later slide on `NetworkSession`.

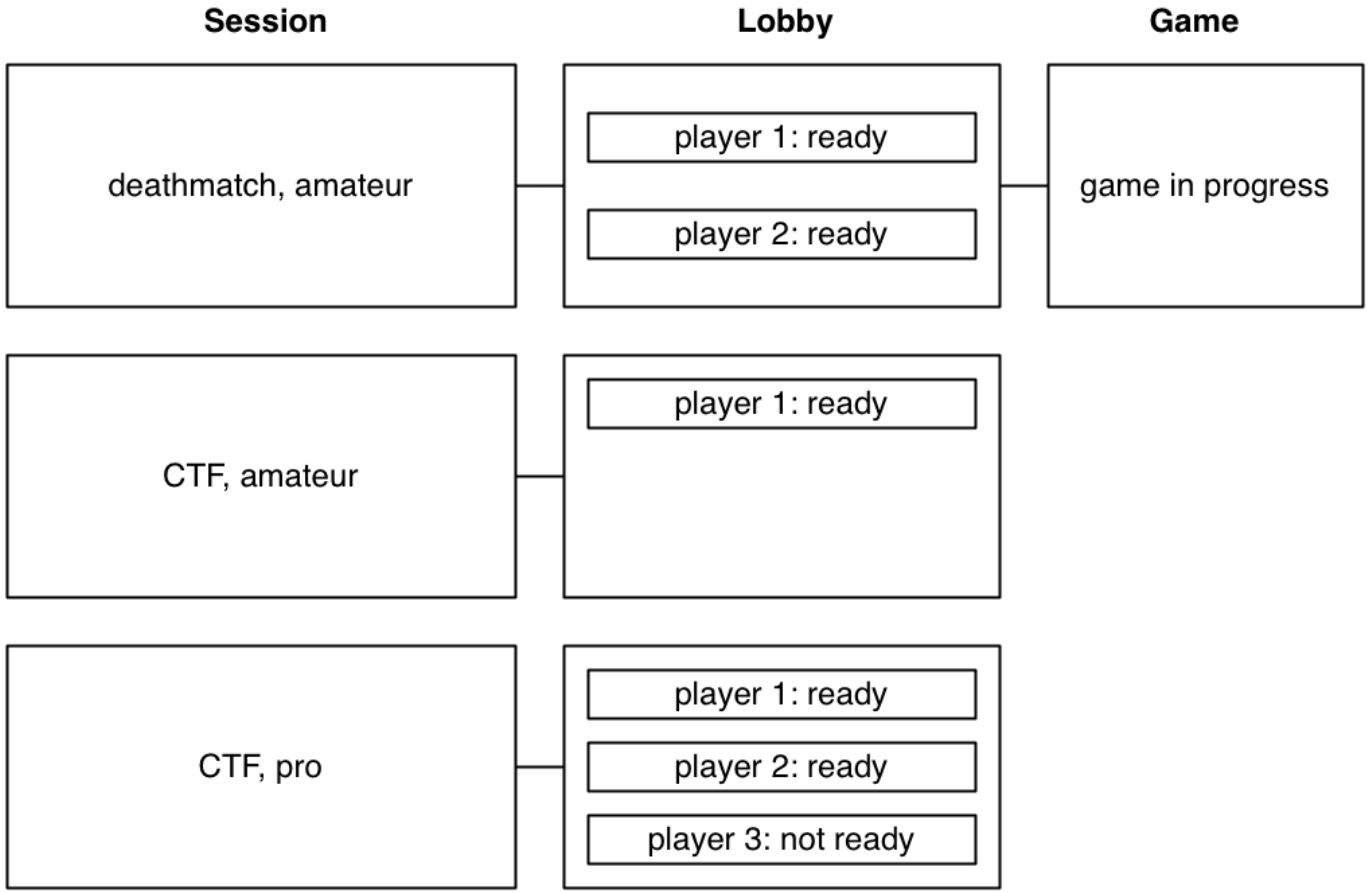
Networked Multi-Player Games

We will illustrate XNA networking using a simple peer-to-peer game. The idea is to present a basic framework that may be extended to incorporate a full game; examples are based on chapter 16 of Miller & Johnson.

The framework works as follows:

- The user can create a new **session**
  - The `NetworkSession` is the backbone of any networked XNA game
  - A session is essentially a set of connections (from different gamers)
  - A session is of a particular type (e.g., capture the flag, deathmatch)
- Alternatively, the user can search for existing sessions
  - The user may then join existing sessions
- Once the session is live, players meet in the **game lobby**
  - Once players joined the lobby, they indicate when they are ready to play
  - The **host** of the session starts the game when all players are ready
  - One or more games may be played in a single session
- The game itself is artificial and simplistic
  - Data is exchanged between players using `PacketWriter` and `PacketReader`

Networked Multi-Player Games



Prerequesites (1/2)

We won't go through all the code required to make this example work and instead focus mostly on the network-related bits of code. The framework we develop has multiple states:

```
1  public enum GameState { MainMenu,
2                          CreateSession,
3                          FindSession,
4                          GameLobby,
5                          PlayingGame };
```

We use 2 additional enumerations for the sessions and games:

```
1  public enum SessionProperties   { GameType, OtherCustomProperty };
2  public enum GameType            { Deathmatch, CaptureTheFlag };
```

We will display timed messages on the screen for user choices and updates:

```
1  public struct DisplayMessage
2  {
3    public string Message;
4    public TimeSpan DisplayTime;
5
6    public DisplayMessage(string message, TimeSpan displayTime)
7    {
8      Message = message;
9      DisplayTime = displayTime;
10   }
11 }
```

Prerequesites (2/2)

For the game itself, we require a game object:

```
1 public class GameObject
2 {
3   public Vector2 Position { get; set; }
4
5   public GameObject(Vector2 position)
6   {
7     Position = position;
8   }
9 }
```

The game allows players to move their gamertags across the screen (to illustrate the exchange of data).

We also need to:

- Keep track of inputs (using gamepads)
- Create and load a font
- Keep a list of timed messages:

```
1 List<DisplayMessage> gameMessages = new List<DisplayMessage>();
```

The framework starts off in `GameState.MainMenu`:

```
1 GameState gameState = GameState.MainMenu;
```

The user then pressed buttons to make choices regarding sessions and games.

Basics of Update() and Draw()

In the `Update()` method, a switch statement checks for the state of the framework and calls the appropriate internal update method. The same mechanism is used in the `Draw()` method (i.e., drawing different menus). The `Draw()` method also draws all timed messaged onto the screen.

The `Update()` method is also used to

- Check the status of messages displayed
- Keep track of user inputs (updating the gamepad states)

We use a helper method called `ButtonPressed()` to register single button presses.

Q Remember how to do this?

The `Update()` method also checks for the number of signed-in users and displays the sign-in dialog if there are none:

```
1 if (Gamer.SignedInGamers.Count == 0 && !Guide.IsVisible)
2   Guide.ShowSignIn(1, false);
```

The `Guide` is the XNA networking dialog where gamer tags can log in: Demo.

Creating a Network Session

The first thing to do is to create a new session. The user can choose this option from the main menu. The framework then enters `GameState.CreateSession`. This offers, via a new menu, the options of `GameType.Deathmatch` and `GameType.CaptureTheFlag`.

We need to create an instance of `NetworkSession` to manage the active session and to send / receive data. The user is the host of this session. The method `CreateSession(GameType gameType)` will be used to initialise the session:

```
1  // If we have an existing network session we need to dispose of it
2  if (networkSession != null && !networkSession.IsDisposed)
3    networkSession.Dispose();
4
5  // Create the NetworkSessionProperties to use for the session
6  // Other players will use these to search for a session
7  NetworkSessionProperties sessionProperties = new
       NetworkSessionProperties();
8  sessionProperties[(int)SessionProperties.GameType] = (int)gameType;
9  sessionProperties[(int)SessionProperties.OtherCustomProperty] = 42;
10
11 // Create the NetworkSession NetworkSessionType of SystemLink
12 networkSession = NetworkSession.Create(NetworkSessionType.SystemLink,
       1, 4, 0, sessionProperties);
13
14 networkSession.AllowJoinInProgress = true;
```

Creating a Session

The method first closes any existing sessions and then sets the properties of the newly created session. The properties

- Can be anything (we used an enumeration for the different attributes)
- These values are used when users search for existing sessions
  - Skill level, map, game type, etc.

The most important line of code is `NetworkSession.Create()`:

```
1 public static NetworkSession Create (
2       NetworkSessionType sessionType,
3       int maxLocalGamers,
4       int maxGamers,
5       int privateGamerSlots,
6       NetworkSessionProperties sessionProperties
7 )
```

- NetworkSessionType: local, localWithLeaderboard, SystemLink, PlayerMatch, Ranked
- maxLocalGamers: number of gamers using same console
- maxGamers: number of total gamers (max is 31)
- privateGamerSlots: slots for invited players (for specific match-ups)
- NetworkSessionProperties: properties of the session

Creating a Session

The command `networkSession.AllowJoinInProgress = true` indicates whether new gamers can join the game / lobby while the game is in progress.

We also need to register a number of useful events in our method:

```
1 networkSession.GameStarted += networkSession_GameStarted;
2 networkSession.GameEnded += networkSession_GameEnded;
3 networkSession.GamerJoined += networkSession_GamerJoined;
4 networkSession.GamerLeft += networkSession_GamerLeft;
5 networkSession.SessionEnded += networkSession_SessionEnded;
6
7 // Move the game into the GameLobby state
8 gameState = GameState.GameLobby;
```

Note:

```
1 networkSession.GameStarted += networkSession_GameStarted;
```

is a shortcut for:

```
1 networkSession.GameStarted += new EventHandler<GameStartedEventArgs>(
      networkSession_GameStarted);
```

These events are triggered when certain things take place to inform the session of changes and to prompt updates. The argument to the `EventHandler` are (very simple) `void` methods we need to define.

EventHandler Methods

Below are some of the methods required for the `EventHandler`:

```
1 void networkSession_GameStarted(object sender, GameStartedEventArgs e)
2 {
3   gameMessages.Add(new DisplayMessage("Game Started", TimeSpan.
        FromSeconds(2)));
4   gameState = GameState.PlayingGame;
5 }
```

```
1 void networkSession_GamerJoined(object sender, GamerJoinedEventArgs e)
2 {
3   gameMessages.Add(new DisplayMessage("Gamer Joined: " + e.Gamer.
        Gamertag, TimeSpan.FromSeconds(2)));
4   e.Gamer.Tag = new GameObject(new Vector2(random.Next(100, 1000),
        random.Next(100, 600)));
5 }
```

Note: `Tag` is not the same as `Gamertag`. It is arbitrary / user-defined data.

```
1 void networkSession_SessionEnded(object sender,
      NetworkSessionEndedEventArgs e)
2 {
3   gameMessages.Add(new DisplayMessage("Session Ended: " + e.EndReason.
        ToString(), TimeSpan.FromSeconds(2)));
4
5   if (networkSession != null && !networkSession.IsDisposed)
6     networkSession.Dispose();
7
8   gameState = GameState.MainMenu;
9 }
```

Game Lobby

The session we just created needs to be updated continuously to be able to communicate with all participants. We call from `Update()`:

```
1 if (networkSession != null && !networkSession.IsDisposed)
2   networkSession.Update();
```

Once the session has been created, the framework enters the game lobby state. The game lobby shows the list of players who joined the session and whether they are ready to play. Each player can indicate that they are ready. The host also has the option to start the game.

Demo.

One can access all the gamers in a sessions as follows:

```
1 foreach (NetworkGamer networkGamer in networkSession.AllGamers)
```

Each `NetworkGamer` has numerous properties such as `Gamertag`, `IsReady` and `IsTalking`.

Playing the Game

When the host decides to start the game, the method `GameLobbyUpdate()` calls

```
1 networkSession.StartGame();
```

This changes the state `NetworkSessionState.Lobby` to `NetworkSessionState.Playing` and triggers the game started event which allows us to update our state to `GameState.PlayingGame`.

The game itself is really simple and allows the gamer to move their gamer tag across the screen. This is just to illustrate the sending and receiving of data. This is done using:

```
1 PacketWriter packetWriter = new PacketWriter();
2 PacketReader packetReader = new PacketReader();
```

We create a new method that updates the game state of the local player, sends that information to all other players (peer-to-peer) and then reads all the position updates from all other players.

Sending and Receiving Data (1/2)

```
1  foreach (LocalNetworkGamer gamer in networkSession.LocalGamers)
2  {
3    // Handle local input
4    GamePadState gamePadState = GamePad.GetState(gamer.SignedInGamer.
         PlayerIndex);
5
6    // Get the GameObject for this local gamer
7    GameObject gameObject = gamer.Tag as GameObject;
8    Vector2 position = gameObject.Position;
9
10   // Update the position of the game object based on the GamePad
11   Vector2 move = gamePadState.ThumbSticks.Left * (float)gameTime.
         ElapsedGameTime.TotalSeconds * 500;
12   move.Y *= -1;
13   position += move;
14   gameObject.Position = position;
15
16   // Send the data to everyone in your session
17   packetWriter.Write(position);
18   gamer.SendData(packetWriter, SendDataOptions.InOrder);
19 }
```

This code does two things:

- Update the gamer's game state according to the left thumbstick
- Send the new position to the other gamers
  - `gamer.SendData()` sends data to a set of gamers in the network session
  - Options include `None`, `Reliable`, `InOrder` and `ReliableInOrder`

Sending and Receiving Data (2/2)

The second part of this method retrieves data from other gamers:

```
1  foreach (LocalNetworkGamer gamer in networkSession.LocalGamers)
2  {
3    // Read until there is no data left
4    while (gamer.IsDataAvailable)
5    {
6      NetworkGamer sender;
7      gamer.ReceiveData(packetReader, out sender);
8
9      // We only need to update the state of non local players
10     if (sender.IsLocal)
11       continue;
12
13     // Get GameObject of the sender
14     GameObject gameObject = sender.Tag as GameObject;
15
16     // Read the position
17     gameObject.Position = packetReader.ReadVector2();
18   }
19 }
```

Loops through all local gamers and receives all the data that has been sent to update their positions. XNA makes is extremely simple to send and retrieve data.

Note: we can ignore data sent from other local gamers as the first loop will have taken care of that.

Searching Available Sessions

The last additions to the framework are to search and join existing sessions. To search for available sessions, we need to store existing sessions as:

```
1 AvailableNetworkSessionCollection availableSessions;
```

We then use `FindSession()` to search for sessions:

```
1 // Define the type of session we want to search for.
2 NetworkSessionProperties sessionProperties = new
      NetworkSessionProperties();
3 sessionProperties[(int)SessionProperties.OtherCustomProperty] = 42;
4
5 // Find an available NetworkSession
6 availableSessions = NetworkSession.Find(NetworkSessionType.SystemLink,
       1, sessionProperties);
7
8 // Move the game into the FindSession state
9 gameState = GameState.FindSession;
```

One does not need to specify all properties. In that case, all matching sessions are returned. The method `NetworkSession.Find()` takes the following arguments:

- Session type
- Number of local players that will be joining the session
- The session properties required

Joining A Session

Once a list of sessions has been obtained, we can join one of them:

```
1  private void JoinSession(int sessionID)
2  {
3    try
4    {
5      networkSession=NetworkSession.Join(availableSessions[sessionID]);
6    }
7    catch (NetworkSessionJoinException ex)
8    {
9      gameMessages.Add(new DisplayMessage("Failed to connect to session:
           " + ex.JoinError.ToString(), TimeSpan.FromSeconds(2)));
10     FindSession();
11   }
12
13   networkSession.GameStarted += networkSession_GameStarted;
14   networkSession.GameEnded += networkSession_GameEnded;
15   networkSession.GamerJoined += networkSession_GamerJoined;
16   networkSession.GamerLeft += networkSession_GamerLeft;
17   networkSession.SessionEnded += networkSession_SessionEnded;
18
19   if (networkSession.SessionState == NetworkSessionState.Playing)
20     gameState = GameState.PlayingGame;
21   else
22     gameState = GameState.GameLobby;
23 }
```

Once connected successfully, the framework moves to the game lobby stage.

Q Why do you think that `Join()` may throw an exception?

More Notes on Networked Games

The quality of service is an important property in networked games. In particular, the *ping* one has can have a great impact on the gameplay experience. An `AvailableNetworkSession` thus has a property called `QualityOfService`.

The `QualityOfService` has properties such as

- `AverageRoundtripTime`: time taken for packets to travel forth and back
- `BytesPerSecondDownstream` and `BytesPerSecondUpstream`: available bandwidth

Delays are unlikely when testing networked games hence it is important to simulate real-world connections. To do so, we can make use of the following:

- `NetworkSession.SimulatedLatency`: gets and set the desired `TimeSpan`
- `NetworkSession.SimulatedPacketLoss`: float between 0 and 1

Q What are games expected to handle in terms of delays and packet loss?

Finally, when the host of a session disconnects, the session is terminated. One can turn on host migration to move the session to another host:

```
1 networkSession.AllowHostMigration = true;
```

2 more demos from the App Hub to look at.

Outline

Assessment

This was the last lecture of the term. Here is a summary of the assessment:

- Progress Test (10%): Done
- Assignment (20%): Due January 17, 2012 (noon)
  - You need to submit project code and report
  - Assignments are to be demonstrated January 18, 2012 (11:00-13:00)
  - This will take place in CES Lab 5; each demo will be approx. 10 minutes
  - Students from the 9am-11am lab: you must attend the entire first hour
  - Students from the 12pm-2pm lab: you must attend the entire second hour
- 2-hour Exam (70%): Summer
  - Might include anything covered in the lectures
  - Essential to consult reading list and web references
  - Important to understand code from labs
  - Revision lecture to take place Friday (15:00-17:00) in week 31

Summary & Lab Preview

In today's lecture, we covered multi-player games. The code will be available on the course website and you should also consult chapter 16 of Miller and Johnson.

- Split-screen multiplayer games
- XNA `Viewport`
- Multiple player objects
- Networked multiplayer games
- Creating a network session
- Game lobby

In the lab you should be working on your assignments. This is the last chance you will have to get help so it is vital you attend the labs to write the code for the foundation of your game.

Q Are there any topics you would like to review?