Join the DarkRift Side

(And together we can rule the galaxy)

# 1 Introduction

Welcome, and it’s nice to know you want to learn DarkRift! This e-book aims to be a comprehensive guide for anyone who wants to get started networking with DarkRift and so I’ll make no assumptions on prior networking experience or knowledge. Nevertheless, as someone who hates oversimplification, I’ll keep it interesting for people who do have networking experience.

If you want to learn DarkRift you’ll need a good grasp of C# though, DarkRift uses multithreading quite extensively so make sure you know the basics! A good knowledge of Unity is also useful since we’ll be using that for the client in this tutorial.

Good luck!

# 2 Install

Create a new Unity project and go to the Unity Asset store, search DarkRift and click download. It will ask you which parts you want to import, just import everything for now.

In the root folder of your project there will now be a zip file entitled DarkRift server, extract it somewhere outside of the Unity project (e.g. your desktop).

Note: If you extract it inside the Unity project Unity will throw a lot errors because there will be library files with the same name in the project so don’t do that.

If you haven’t already, now’s a great time to have a look at the demo projects included in DarkRift and get a feel for how to use the server.

# 3 Basics

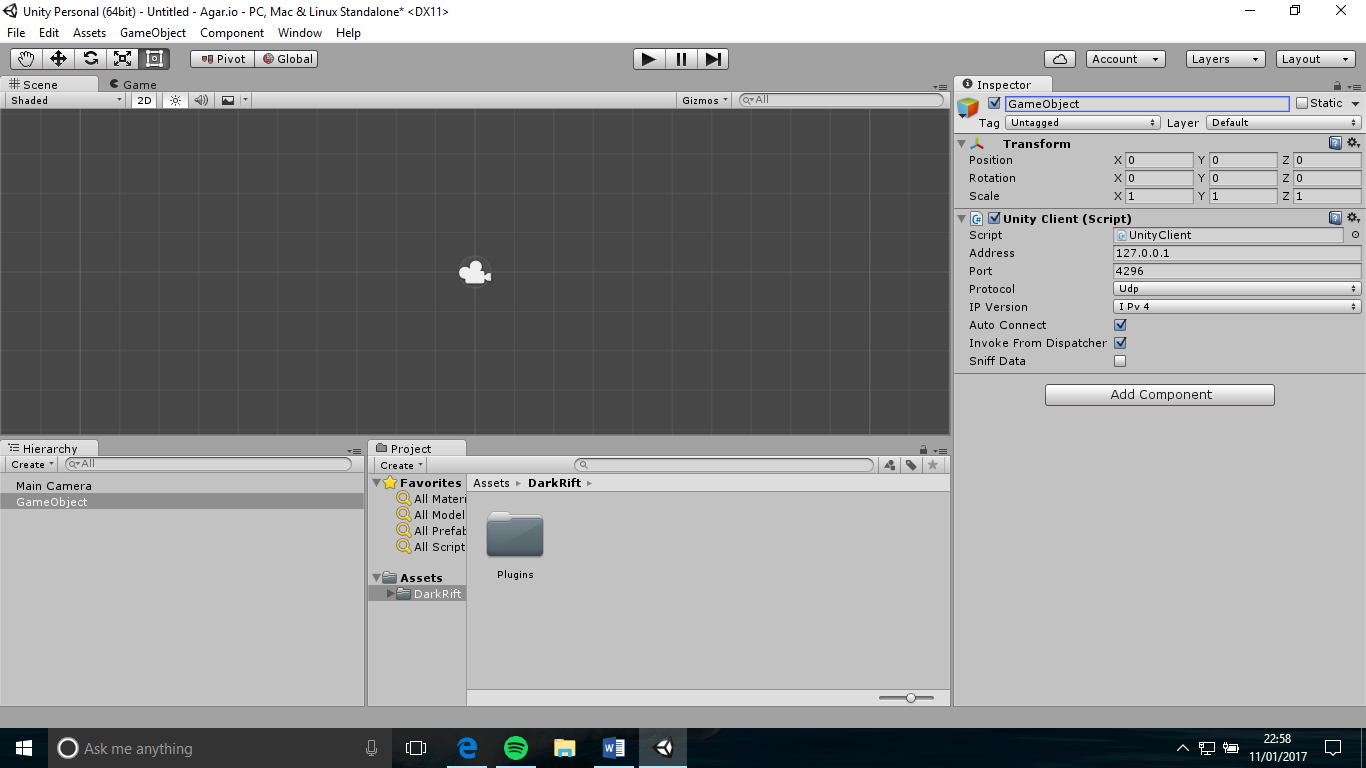
As a tutorial, we’re going to recreate a popular game called [Agar.io](http://agar.io). We won’t implement all the features it has but we’ll get a basic system of moving and eating going and you can always add the others as an exercise afterwards.

## Unity Scene

### Setting Up DarkRift

In a blank Unity scene add an empty GameObject, rename it Network and then add a DarkRift -> Client component.

This is the first fundamental DarkRift part, the client script connects to the server for you and allows you to send and receive messages - plus it even has some useful tools built in for you! Let’s look at the Unity interface to it.



* *Address* is the host IP address that the server you wish to connect to is located on, by default it’s set to local host (the same computer you’re running the game on) for development.
* *Port* is the port number you’ve set the server to listen on, by default it’s 4296.
* *Protocol* is the networking protocol you’d like DarkRift to connect to the server using, this needs to be the same as the server is listening on. We’ll discuss these later.
* *IP version* is the type of IP address that you want to connect to the server with, this is most likely IPv4.
* *Auto Connect* means that the client will try and establish a connection when Unity invokes the Start method. If you turn this off, you’ll need to manually call UnityClient.Connect() from your code.
* *Invoke From Dispatcher* means the client will raise all events from Unity’s main thread so that you don’t need to worry about synchronization and multi-threading. If you disable this, you’ll need to handle synchronization and dispatching to the main thread yourself but you might see performance benefits. See Dispatcher.
* *Sniff Data* means that the client will print the details of messages received for debugging purposes.

Leave everything as is.

### Setting up the Scene

Because we’re making a 2D game make sure your Scene view is in 2D mode and make the camera orthographic.

Set the camera’s background colour to a white colour.

### Players

Download or make a white circle with a transparent background so we can colour it for different players, the higher the resolution the better. [I used this big white circle](https://goo.gl/images/hK8leu).

Create a C# script called AgarObject.cs and add the following helper functions that we’ll come to use later:

[RequireComponent(typeof(Renderer))]

public class AgarObject : MonoBehaviour

{

[SerializeField]

[Tooltip("The speed that the player will move.")]

float speed = 1f;

[SerializeField]

[Tooltip("Multiplier for the scaling of the player.")]

float scale = 1f;

Vector3 movePosition;

void Awake()

{

movePosition = transform.position;

}

void Update()

{

if (speed != 0f)

transform.position = Vector3.MoveTowards(transform.position, movePosition, speed \* Time.deltaTime);

}

internal void SetColor(Color32 color)

{

Renderer renderer = GetComponent<Renderer>();

renderer.material.color = color;

}

internal void SetRadius(float radius)

{

transform.localScale = new Vector3(radius \* scale, radius \* scale, 1);

}

internal void SetMovePosition(Vector3 newPosition)

{

movePosition = newPosition;

}

}

Add that and the big circle to an empty game object and make it a prefab called NetworkPlayer then create a new C# script called MouseController and add the following code to the update routine:

[RequireComponent(typeof(AgarObject))]

public class MouseController : MonoBehaviour

{

AgarObject agarObject;

void Awake()

{

agarObject = GetComponent<AgarObject>();

}

void Update ()

{

Vector3 mousePoint = Camera.main.ScreenToWorldPoint(Input.mousePosition);

mousePoint.z = 0;

agarObject.SetMovePosition(mousePoint);

}

}

Add the MouseController script to the player game object and make it a separate prefab called ControllablePlayer (for clarity you should now have 2 versions: NetworkPlayer without MouseController and ControllablePlayer with MouseController). You should be able to test your player and get a feel for the right speed value for you.

Make sure the speed on the NetworkPlayer is maybe 1.5x the speed on the ControllablePlayer.

### Food

Download or make a food-like white square with a transparent background so we can give it different colours. Since it’s a square the transparent background might not be necessary. I used this [big white square](https://goo.gl/images/lOA64v).

Make a new prefab consisting only of the square and the AgarObject script, set the speed of it to 0 and then ensure your scene only consists of our Network object and Main Camera. Now is also a good time to save and backup your work/commit if you haven’t done so yet.

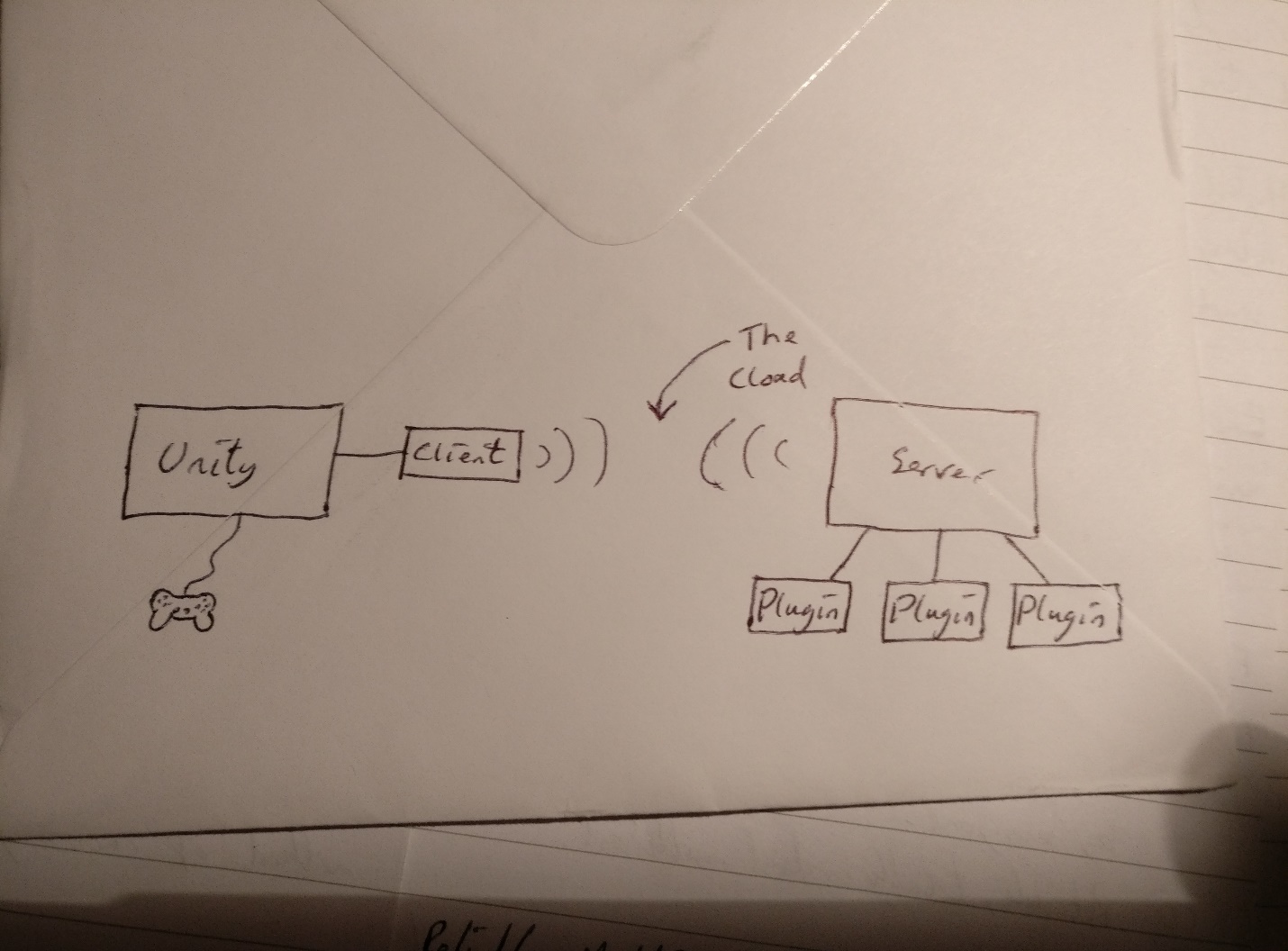
And that’s all the client side logic we’re going to do for now!

## Server Basics

### A Brief Overview

Before we jump into the art of DarkRift we should explore how DarkRift operates.

DarkRift uses plugins to implement all server side logic, you write plugins as you would develop a standard .NET library and then you just drop the generated DLL into DarkRift’s ‘Plugins’ directory. DarkRift offers a large API for managing clients and sending messages etc. and provides events that inform your plugins when a client connects, sends a message etc. See the appalling diagram written on the back of an envelope:



Everything related to the server code is in the namespace DarkRift.Server, everything to do with the client in DarkRift.Client and everything used in both is in DarkRift.

### Setting up the Plugin

Create a new Class Library C# project in Visual Studio named something like “AgarPlugin” and add references to DarkRift.Server.dll and DarkRift.dll.

Rename the default class to AgarPlayerManager and rename the file similarly. Add using directives to the top of the file so that we have access to both DarkRift.Server and DarkRift namespaces.

All plugins in DarkRift must inherit from DarkRift.Server.Plugin so that DarkRift can correctly discover and instantiate them and so it can easily read details about your plugin. It is also your entry point into DarkRift’s API so that you make changes on the correct server instances.

Go ahead and make AgarPluginManager inherit from Plugin, you’ll notice Visual Studio throws an error saying that there is no constructor so click “Quick Actions and Refactorings” (or the little lightbulb) and add it. You’ll then notice it says that it does not implement a number of abstract properties, once again add them. If you’re following along, you should have a class like this:

public class AgarPlayerManager : Plugin

{

public AgarPlayerManager(PluginLoadData pluginLoadData) : base(pluginLoadData)

{

}

public override string Name

{

get

{

throw new NotImplementedException();

}

}

public override bool ThreadSafe

{

get

{

throw new NotImplementedException();

}

}

public override Version Version

{

get

{

throw new NotImplementedException();

}

}

}

Hopefully the autogenerated properties should be fairly self-explanatory. When implementing them remember:

* Name should be something that is obvious since it will appear on all associated messages and traces.
* Version should follow [Semantic Versioning](http://semver.org/) (unless you really hate best practices). It is used to trigger upgrades to plugins (we’ll cover that later) so you should give it a default value now.
* ThreadSafe indicates whether your program can handle multithreaded events to boost performance. Since I won’t cover any multithreading in this set it to false, if you know about multithreading then it might be a nice exercise at the end to reimplement the plugin with multithreading enabled!

After some formatting, we should have something like:

public class AgarPlayerManager : Plugin

{

public override string Name => nameof(AgarPlayerManager);

public override bool ThreadSafe => false;

public override Version Version => new Version(1, 0, 0);

public AgarPlayerManager(PluginLoadData pluginLoadData) : base(pluginLoadData)

{

}

}

If you want to try it you can build the library and place the DLL in DarkRift’s plugins folder, when you start DarkRift you should see your plugin get loaded in!

## First Network Code

### Spawning Players

The first thing that we need to do is make players appear when someone logs in. We need to make sure that the person logging in receives a list of all players that are currently online and we need to make sure everyone else spawns our new player on their game.

Let’s take a step back though and plan what we need to store about a player:

* Their X, Y position in the world.
* Their radius (when they eat someone they’re going to get better)
* Their colour.
* Plus, in the future we might want to store a name.

Go ahead and make a new Player object on the server that stores those. If you’re lazy like me, copy and paste this code in:

class Player

{

public uint ID { get; set; }

public float X { get; set; }

public float Y { get; set; }

public float Radius { get; set; }

public byte ColorR { get; set; }

public byte ColorG { get; set; }

public byte ColorB { get; set; }

public Player(uint ID, float x, float y, float radius, byte colorR, byte colorG, byte colorB)

{

this.ID = ID;

this.X = x;

this.Y = y;

this.Radius = radius;

this.ColorR = colorR;

this.ColorG = colorG;

this.ColorB = colorB;

}

}

I use bytes for the color because it means we have to store and send a lot less data that when using floats and it will serve the exact same purpose. Taking little steps like this can really improve bandwidth when you sending a message multiple times.

To spawn players we need to tell DarkRift to inform us when a player connects. In the constructor of AgarPlayerManager add the line:

ClientManager.ClientConnected += ClientConnected;

And add a new method:

void ClientConnected(object sender, ClientConnectedEventArgs e)

{

}

The ClientManager is responsible for keeping track of any clients that are connected to the server so you can use it to access clients and it can inform you when clients connect or disconnect. In this line, we’re subscribing to the ClientConnected event so whenever someone connects our method will be called with the ClientManager in sender and details of the connection in the ClientConnectedArgs.

Let’s give our newly connected player their own player object. In our ClientConnected method add code to generate them a new player setup with random values:

Random r = new Random();

Player newPlayer = new Player(

e.Client.GlobalID,

(float)r.NextDouble() \* MAP\_WIDTH - MAP\_WIDTH / 2,

(float)r.NextDouble() \* MAP\_WIDTH - MAP\_WIDTH / 2,

1f,

(byte)r.Next(0, 200),

(byte)r.Next(0, 200),

(byte)r.Next(0, 200)

);

And at the top of the class add a MAP\_WIDTH constant:

const float MAP\_WIDTH = 20;

You might notice the sneaky e.Client.GlobablID hiding in there. So that we can address the player when it’s on the Unity side we need to have an identifier, since the client will only have one player object they control it’s sufficient to just use the client’s global ID that gets allocated to it by DarkRift when it connects.

I’m using a value between 0 and 200 for each color channel because if it’s too close to 255, 255, 255 we won’t be able to see the client against the background!

Next let’s send our new Player object to all the other clients so that they can spawn a new player in their games. Add the following just after:

DarkRiftWriter newPlayerWriter = new DarkRiftWriter();

newPlayerWriter.Write(newPlayer.ID);

newPlayerWriter.Write(newPlayer.X);

newPlayerWriter.Write(newPlayer.Y);

newPlayerWriter.Write(newPlayer.Radius);

newPlayerWriter.Write(newPlayer.ColorR);

newPlayerWriter.Write(newPlayer.ColorG);

newPlayerWriter.Write(newPlayer.ColorB);

Message newPlayerMessage = new TagSubjectMessage(0, 0, newPlayerWriter);

foreach (Client client in ClientManager.GetAllClients().Where(x => x != e.Client))

client.SendMessage(newPlayerMessage, SendMode.Reliable);

Whoa there! That got complicated quickly didn’t it…!

Let’s look through each part. We need to convert our player to something we can send over the network and currently it’s a unsigned integer, 3 floats and 3 bytes which isn’t compatible with the internet. The DarkRiftWriter and DarkRiftReader are objects provided for you to make that conversion easier, we write all our fields into the writer and on the Unity side we’ll read them off **in the same order** with a reader.

Next, we construct a TagSubjectMessage. Messages essentially wrap data (all those values we stuffed into the DarkRiftWriter) and give them some simple header values that identify what the data actually contains; TagSubjectMessages contain 2 headers, the tag and the subject. The tag is a single byte that identifies what subsystem the message should be delivered to (think WorldManager, PlayerManager, ChatManager, PlayerSpawner) and the subject is a single ushort that identifies to that subsystem what the message is about.

Finally, we get all clients currently connected to the server using ClientManager.GetAllClients(), utilise a little LINQ to remove the client that just connected (we’ll deal with him later) and then send the message to each client. Don’t forgot that when you call GetAllClients the client that just connected has already been added!

See? Easy! What were you worried about?

In all seriousness though, that’s pretty much 80% of everything there is to sending messages in DarkRift, both client and server: create a writer for the data, a message to wrap the data and then send.

One thing I did gloss over, however, was the SendMode. This identifies how we should send the message to the client:

* Unreliable – No guarantees will be made that the client gets the message but it’s a lot faster to send. Use this for frequent data that will soon be out of data anyway (e.g. position/rotation updates).
* Reliable – The client is guaranteed to receive this message at the expense of needing to acknowledge back to the sender it has got it. Use this for important data that cannot be lost (e.g. chat messages or inventory updates).
* FragmentedReliable – Like reliable but large messages will be split up into smaller messages so that they can be sent. Use this for anything that might be overly big (e.g. pictures).

The next thing we need to do quickly is to spawn all other players on our newly connected player; since we want to minimise the bandwidth we use here we’re going to package it all into a single message and send it. Add the following line to the top of your AgarPlayerManager:

Dictionary<Client, Player> players = new Dictionary<Client, Player>();

And then add this to the end of your ClientConnected method:

players.Add(e.Client, newPlayer);

DarkRiftWriter playerWriter = new DarkRiftWriter();

foreach (Player player in players.Values)

{

playerWriter.Write(player.ID);

playerWriter.Write(player.X);

playerWriter.Write(player.Y);

playerWriter.Write(player.Radius);

playerWriter.Write(player.ColorR);

playerWriter.Write(player.ColorG);

playerWriter.Write(player.ColorB);

}

Message playerMessage = new TagSubjectMessage(0, 0, playerWriter);

e.Client.SendMessage(playerMessage, SendMode.Reliable);

Hopefully this should be fairly self-explanatory from what we’ve already done, the only difference is that we keep writing into the same writer so that it’s all in one package. Note that we add the new player to the dictionary before enumerating over it so that it’s included in what we send to the player (otherwise they wouldn’t spawn themselves a player!)

Finally, it’s a good idea to use define constants for tags and subjects so that you can easily refer to them and modify them without side effects. Add the following declaration at the top:

const byte SPAWN\_TAG = 0;

And change the tag parameter in **both** constructor calls for the TagSubjectMessage:

new TagSubjectMessage(SPAWN\_TAG, 0, writer);

### Actually Spawning Players

Now that we’ve written the server code for spawning players let’s add the client side code. Create a new file called PlayerSpawner.cs in the Unity project and add the following code:

public class PlayerSpawner : MonoBehaviour

{

const byte SPAWN\_TAG = 0;

[SerializeField]

[Tooltip("The DarkRift client to communicate on.")]

UnityClient client;

[SerializeField]

[Tooltip("The controlable player prefab.")]

GameObject controllablePrefab;

[SerializeField]

[Tooltip("The network controlable player prefab.")]

GameObject networkPrefab;

void Awake()

{

if (client == null)

{

Debug.LogError("Client unassigned in PlayerSpawner.");

Application.Quit();

}

if (controllablePrefab == null)

{

Debug.LogError("Controllable Prefab unassigned in PlayerSpawner.");

Application.Quit();

}

if (networkPrefab == null)

{

Debug.LogError("Network Prefab unassigned in PlayerSpawner.");

Application.Quit();

}

client.Subscribe(SPAWN\_TAG, SpawnPlayer);

}

}

You’ll need to add a reference to DarkRift.Client.Unity as well.

In this you’ll notice we define a reference to a UnityClient object that we’ll fill from the inspector. This is the component we added to the Network object in our Unity scene earlier and handles our connection to the server, remember?

When we call subscribe on it we tell it that we are the designated handler for all messages with SPAWN\_TAG in its tag field and hence those messages, and only those messages, should be sent to us for processing. Be aware that only one handler at a time can be subscribed to a given tag in this way and re-subscribing will overwrite prior subscriptions!

People who used DarkRift 1 will be used to receiving all messages in a single event, this is still available as it’s needed for some other purposes like receiving SimpleMessages but I recommend you transition to this system in time because it allows DarkRift to make some extra optimisations for you (and promotes better design, not that your design is bad…).

Add the following code to decode our spawn packets:

void SpawnPlayer(object sender, MessageReceivedEventArgs e)

{

DarkRiftReader reader = e.Message.GetReader();

if (reader.Length % 19 != 0)

{

Debug.LogWarning("Received malformed spawn packet.");

return;

}

while (reader.Position < reader.Length)

{

uint id = reader.ReadUInt32();

Vector3 position = new Vector3(reader.ReadSingle(), reader.ReadSingle());

float radius = reader.ReadSingle();

Color32 color = new Color32(

reader.ReadByte(),

reader.ReadByte(),

reader.ReadByte(),

255

);

GameObject obj;

if (id == client.ID)

{

obj = Instantiate(controllablePrefab, position, Quaternion.identity) as GameObject;

}

else

{

obj = Instantiate(networkPrefab, position, Quaternion.identity) as GameObject;

}

Player agarObj = obj.GetComponent<Player>();

agarObj.SetRadius(radius);

agarObj.SetColor(color);

}

}

If you take a careful look into the code you’ll see that we just reverse the packaging process we did before: we get a reader from the message that contains the message data and simply loop through reading the data out in the same order we wrote it on (note, order matters here!) until there is no more data left. As we read through we create the necessary objects depending on whether the ID of the player object is our ID (if it is then that should be the object we’re controlling).

Add the PlayerSpawner component to the Network object in the scene and drag onto it the client and both prefabs in their respective places.

You should now be able to test and see a player spawn on the client. Next step, movement!

## Movement!

### Sending our position

In the Unity project create a new C# script called player, this is going to send our position to the server every time we move. Add the following code to it:

public class Player : MonoBehaviour

{

const byte MOVEMENT\_TAG = 1;

const ushort MOVE\_SUBJECT = 0;

[SerializeField]

[Tooltip("The distance we can move before we send a position update.")]

float moveDistance = 0.05f;

public UnityClient Client { get; set; }

Vector3 lastPosition;

void Awake()

{

lastPosition = transform.position;

}

void Update()

{

if (Vector3.Distance(lastPosition, transform.position) > moveDistance)

{

/\* Send position to server here \*/

lastPosition = transform.position;

}

}

}

Add it onto your ControllablePlayer prefab. In our PlayerSpawner change our if statement in the while loop to:

if (id == client.ID)

{

obj = Instantiate(controllablePrefab, position, Quaternion.identity) as GameObject;

Player player = obj.GetComponent<Player>();

player.Client = client;

}

else

{

obj = Instantiate(networkPrefab, position, Quaternion.identity) as GameObject;

}

Now that’s done try to fill in the Player class with code to send our position. All that needs to be sent is the x and y components of transform.position.

If you’re stuck, here’s the code I wrote:

DarkRiftWriter writer = new DarkRiftWriter();

writer.Write(transform.position.x);

writer.Write(transform.position.y);

TagSubjectMessage message = new TagSubjectMessage(MOVEMENT\_TAG, MOVE\_SUBJECT, writer);

Client.SendMessage(message, SendMode.Unreliable);

None of this should look unfamiliar, we create a writer and package the data we want to send, put that inside a message with the movement tag and then send unreliably (if the server doesn’t get the position it’s not the end of the world).

### Updating the Server Position

On our server we need to track the positional changes of each player so that new players connecting always get the latest position, we also need to send this movement update message out to all other players.

In our server plugin append the following line to the ClientConnected method:

e.Client.Subscribe(MOVEMENT\_TAG, MovementMessageReceived);

You’ll need to also add the movement tag constant to the file (it was set to 1).

Hopefully this line should look similar to what we did on the client, we’re telling the client object that whenever it receives a Movement message it should forward it to our handler. Let’s add that handler now:

void MovementMessageReceived(object sender, MessageReceivedEventArgs e)

{

DarkRiftReader reader = e.Message.GetReader();

float newX = reader.ReadSingle();

float newY = reader.ReadSingle();

Client client = (Client)sender;

Player player = players[client];

player.X = newX;

player.Y = newY;

DarkRiftWriter writer = new DarkRiftWriter();

writer.Write(player.ID);

writer.Write(player.X);

writer.Write(player.Y);

writer.Write(player.Radius);

e.Message.SetWriter(writer);

e.DistributeTo.UnionWith(ClientManager.GetAllClients().Where(x => x != client));

}

Once again, the first few lines of this should look familiar. The writer here is just updating the contents of the message so we include the player ID (we need to know who sent it when we receive on the client) and we’re including the radius here for later. But what is that last line doing?

Answer: Nothing special…

MessageReceiveEventArgs contains 2 shortcuts, the first is the DistributeTo hash set which is just a list of people that the message will be sent to when the event handler exits. This means that you don’t have to manually send the message out each time and that if multiple plugins handle the same tag they’re less likely to send duplicate messages out. DistrubuteTo is always initialised empty.

The second shortcut is the ShouldDistribute boolean, if you don’t want something distributed set this to false. You shouldn’t ever have to set this to true because you could override another plugin’s request to not distribute and this could cause conflict issues. This is always initialised as true.

### Updating the Client Position

Since we only want one script handling each tag we’re going to move all clients with a manager. Create a new C# script in your Unity project called NetworkPlayerManager and add the following:

public class NetworkPlayerManager : MonoBehaviour

{

const byte MOVEMENT\_TAG = 1;

[SerializeField]

[Tooltip("The DarkRift client to communicate on.")]

UnityClient client;

Dictionary<uint, AgarObject> networkPlayers = new Dictionary<uint, AgarObject>();

public void Add(uint id, AgarObject player)

{

networkPlayers.Add(id, player);

}

}

Add the script to our Network object and assign the client field. Add the following field to the Player Spawner and assign it in the inspector:

[SerializeField]

[Tooltip("The network player manager.")]

NetworkPlayerManager networkPlayerManager;

Finally add this line to the bottom of your SpawnPlayer method:

networkPlayerManager.Add(id, agarObj);

Have a go at writing the logic for this yourself, don’t forget you can look back at the receive code in PlayerSpawner and the plugin code we wrote for reference. If you get stuck, here’s my implementation:

public void Start()

{

client.Subscribe(MOVEMENT\_TAG, MovementMessageReceived);

}

void MovementMessageReceived(object sender, MessageReceivedEventArgs e)

{

DarkRiftReader reader = e.Message.GetReader();

uint id = reader.ReadUInt32();

Vector3 newPosition = new Vector3(reader.ReadSingle(), reader.ReadSingle(), 0);

networkPlayers[id].SetMovePosition(newPosition);

networkPlayers[id].SetRadius(reader.ReadSingle());

}

Go into Edit -> Project Settings -> Player and check “Run in Background”.

Build it and test it! We should have movement!

## The Sniffer

Let’s take some time out from the coding to look at one of DarkRift’s most powerful debugging features.

Currently DarkRift is just sitting there, occasionally telling you when someone connects or disconnects, what if I want to know if it actually receives a message I send?

Restart your server and type into it:

**sniff -a**

Now when you test you should see every message that enters the server get printed to the console window!

Since that’s a mess of information lets narrow it down. Firstly though, you can use the **-**r flag to remove a previously sniffed rule:

**sniff -a -r**

There are a number of filters you can use with the sniffer:

|  |  |  |
| --- | --- | --- |
| **Filter** | Description | Example |
| **-t** | Sniffs for a specified tag | -t=9 |
| **-s** | Sniffs for a specified subject | -s=80 |
| **-ip** | Sniffs for a specified IP address | -ip=10.56.81.2 |
| **-id** | Sniffs for a specified client | -id = 10 |

You can specify multiple filters at a time and they will logically AND together (i.e. a message has to satisfy both filters to be printed) and you can specify multiple sniffing rules at a time and they will logically OR (i.e. a message will be printed if it satisfies any defined sniffing rule).

Have a play with it in some of the example scenes; you probably won’t be able to have quite as much fun with it in our tutorial game since we only ever send one tag and subject from the clients!

For those of you who knew the original system in DarkRift 1 hopefully you’ll appreciate how much better this is!

## Disconnections

You’ve probably noticed by now that players don’t disappear when they exit the game, lets jump back into the code for a bit and resolve that.

### Disconnection Events

We can get the server to tell us when clients disconnect in the same way we get it to tell us when they connect. Add the following to the constructor of the AgarPlayerManager plugin:

ClientManager.ClientDisconnected += ClientDisconnected;

I hope you can see the similarities! Now add a method that will clean up and inform all our remaining players:

void ClientDisconnected(object sender, ClientDisconnectedEventArgs e)

{

players.Remove(e.Client);

DarkRiftWriter writer = new DarkRiftWriter();

writer.Write(e.Client.GlobalID);

TagSubjectMessage message = new TagSubjectMessage(SPAWN\_TAG, 1, writer);

foreach (Client client in ClientManager.GetAllClients())

client.SendMessage(message, SendMode.Reliable);

}

Note that we’re now using the subject field to differentiate between spawning and de-spawning so go ahead and add constants for these.

### Removing on Unity

Once again this is nothing new, go ahead and try to implement it yourself. For this I restructured some of PlayerSpawner but, as always, my solution was to add:

void SpawnTagHandler(object sender, MessageReceivedEventArgs e)

{

TagSubjectMessage message = (TagSubjectMessage)e.Message;

if (message.Subject == SPAWN\_SUBJECT)

SpawnPlayer(sender, e);

else

DespawnPlayer(sender, e);

}

void DespawnPlayer(object sender, MessageReceivedEventArgs e)

{

DarkRiftReader reader = e.Message.GetReader();

networkPlayerManager.DestroyPlayer(reader.ReadUInt32());

}

And to change the subscribe call to:

client.Subscribe(SPAWN\_TAG, SpawnTagHandler);

Finally, I added a DestroyPlayer method to our NetworkPlayerManager:

public void DestroyPlayer(uint id)

{

AgarObject o = networkPlayers[id];

Destroy(o.gameObject);

networkPlayers.Remove(id);

}

Hopefully you’re now getting the hang of it and feel you can implement new features on your own. Once you know where things are in DarkRift it’s pretty much just the same principles replayed over!

## The Configuration File

Let’s have another moment out to look through the configuration file. The configuration file holds lots of settings that DarkRift uses to start up including where to look for plugins, what to log and how to operate on the network.

Open the Server.config file that’s in the same directory as the server executable.

Firstly, you should know that the name of this file is one of the few hardcoded things in DarkRift, if you change it DarkRift will throw an error and won’t be able to start. The second thing to know is that any settings changed in the server configuration won’t be reflected in DarkRift until you restart the server.

The server tag defines how DarkRift operates on the network, for example if you wanted to change to TCP communication then you’d find that setting here.

The data tag defines how DarkRift should store persistent data, there’s more on this in a later chapter but for now you just need to know that DarkRift stores the databases for itself (specified by the localDB attribute) and your plugins in the directory specified here. The data directory is hidden by default in case you’re looking for it!

The logging tag instructs DarkRift how to log output to the various channels available to it. It should be fairly obvious what these do from their names so if you want to reroute certain log levels to different outputs then you can edit the rules here.

The plugins tag defines where DarkRift should look to find plugins, you can disable the automatic directory loading and change the location that it looks in. You can also specify specific plugin libraries to load if you want to filter out different functionality etc. Go ahead and disable the automatic loading if you like, you can add the following to ensure it still loads our Agar plugin as well:

<plugins loadFromDirectory="false" pluginDirectory="Plugins">

<plugin src="Plugins/AgarPlugin.dll" />

</plugins>

Lastly, the databases tag allows you to specify database connection strings for your plugins to retrieve later so they can connect to remote storage facilities.

## Omnomnomnom

Let’s eat. I feel like our strange circles floating around their empty abyss are probably getting hungry by now…

For this I’m deliberately going to leave out simple code snippets since you should now have a good enough understanding to be able to add the easy bits yourself. If you’re struggling, go back and look through the previous chapters and the code you’ve already written and, if all else fails, look it up in the complete project.

Add a new C# file to our server plugin called “AgarFoodManager” and make the class a DarkRift plugin (first challenge!). Make sure you set ThreadSafe to false so you don’t have problems later!

Add a HashSet field called food that contains FoodItem objects and initialise it empty. Add the following definition of FoodItem to a new file:

public class FoodItem : IEdible

{

public uint ID { get; set; }

public float Radius => 0.1f;

public float X { get; set; }

public float Y { get; set; }

public byte ColorR { get; set; }

public byte ColorG { get; set; }

public byte ColorB { get; set; }

}

Now add the following definition of IEdible to a new file and make Player implement the IEdible interface (see where this is going?).

public interface IEdible

{

uint ID { get; }

float X { get; }

float Y { get; }

float Radius { get; }

}

In the food manager constructor add a number of food items (I did 20) with random values to the food HashSet similarly to how we assigned values to the players. For the ID field just assign them incrementing values from 0 so we can identify then uniquely later.

Now for a more challenging part! Make it so that when a player connects to the server the AgarFoodManager sends the latest positions of all food to the new client, ideally this should be in one message as we did before with the players (hint) but if you’re struggling just implement it as 1 message per food item and get that working first. Use a dedicated food tag and a spawn subject for this since we’ll be doing more with this tag later.

Switching to the Unity project, add a FoodManager MonoBehaviour and attach it to the Network object. In this script, you’ll need to pass in the client and receive the list of food that the server sends, again you can probably reuse some of the player code here. Make sure you track the food items you add using a dictionary (ID as key) so that we can access them later.

You should be able to test now and see lots of food appear on the map with your players!

While we’re testing, add some code to make the camera in Unity a child of our player so it follows him round!

Let’s switch back to AgarPlayerManager and add in some code to actually eat the items of food! In our MovementMessageReceived method add the following code:

player.X = newX;

player.Y = newY;

AgarFoodManager foodManager = PluginManager.GetPluginByType<AgarFoodManager>();

foreach (FoodItem food in foodManager.Food)

{

if (Math.Pow(player.X - food.X, 2) + Math.Pow(player.Y - food.Y, 2) < Math.Pow(player.Radius, 2))

{

player.Radius += food.Radius;

SendRadiusUpdate(player);

foodManager.Eat(food);

}

}

It should be fairly easy to make out the Pythagoras in there. Essentially, we’re just checking the distance between the food and the player is less than the player’s size in a makeshift collision routine, we can then increase our player’s size appropriately and “eat” the food.

This won’t compile yet because there’s three missing items; before we add them, let’s look at the GetPluginsByType call. Since DarkRift handles the instantiation of plugins you can’t get a reference to other plugins without querying DarkRift because DarkRift is the entity with that reference so that’s exactly what we do here.

Hopefully you can see that the Food property and the Eat method need to be implemented in the AgarFoodManager class so add the following line:

public IEnumerable<FoodItem> Food => food;

The SendRadiusUpdate function should send a message containing the new radius to every client using the movement tag but with a different subject, you’ll also need to check the subject you receive for movement messages in NetworkPlayerManager on the client and either call SetMovePosition or SetRadius on the AgarObject so it does the right action!

Lastly, have a go at implementing the Eat method. All this needs to do is move the item of food to another, random location and inform all connected clients of this. Add yourself a MOVE\_SUBJECT constant for issuing the move command.

The only thing that needs adding on the client now is some code to handle the movement of food items, everything else is already handled by the server side logic! Have a go at adding it yourself.

If you have trouble with the collisions not lining up with your graphics try adjusting the scale value on the AgarObjects.

## Implementing Cannibalism

Since we’ve built the eating system generic it’s easy to let other players get eaten. Add the following just after the food eating foreach loop:

foreach (Player p in players.Values)

{

if (p != player && Math.Pow(player.X - p.X, 2) + Math.Pow(player.Y - p.Y, 2) < Math.Pow(player.Radius, 2))

{

player.Radius += p.Radius;

SendRadiusUpdate(player);

Kill(p);

}

}

I’ll let you implement Kill how you see fit, it’s probably best to send a ‘killed’ message to the player so that they can show a game over screen.

And that’s it! Our game is complete. If you want to continue, here are some suggestions for extensions or alternatively you can go onto the advanced section if you want more DarkRift!

* Full GUI
* Restart game once killed
* Animate the players (e.g. expand to engulf food, grow in size slowly when eating, etc.)
* Reduce speed as size increases
* Ability to shed weight to get a speed boost
* Pickups (e.g. invulnerability, speed boost, size boost, etc.)

# 4 Advanced Plugins

## Architecture

Now, I’m afraid that’s most the practical DarkRift stuff out the way. Whilst there will still be practical elements there’s a lot more theory in this second part. Nevertheless, stick with it because it’s important to know.

### Server Architecture

Right at the beginning you saw a diagram showing how DarkRift was very roughly composed, hopefully that gave you a good basic overview of how the server is structured, however, it’s a little more complicated than that for certain things to work.

Firstly, above the DarkRift server is a host process. DarkRift is no more than a library that can be embedded into things and so something must act as an overseer that can start it and watch it. For the purposes of this tutorial we’ve been using the Console Server which is essentially a small program that starts a server and pumps anything on the standard input stream as a command into DarkRift. Another popular choice of host process is a Unity project, doing this allows the server code to utilise Unity’s physics, navmesh and much more in order to simulate the players within game worlds.

Secondly, we mentioned the sniffer earlier but never touched on how it was implemented. Within DarkRift there are several internal plugins embedded that provide helpful functions like, development aids or overseers. Having these as plugins makes it much simpler to handle things like command input, could allow for more modular updates in the future and are generally a good idea. In fact, anything you can input commands into are implemented as plugins (although a couple of them have special privileges)!

Lastly, you probably saw right at the beginning that DarkRift said “Installed plugin AgarPlayerManager version 1.0.0” and completely dismissed it. We also briefly mentioned it earlier but DarkRift has a local data store in the Data folder (by default) to which is a hidden folder if you’re looking for it. This contains several SQLite databases containing information relating to plugins etc. and individual databases for your plugins to store in as needed (we’ll come to this later). Word of warning, don’t change the contents of these because it’ll be a nightmare to fix but it may be a good idea to include this folder in any automatic backup systems you have!

### Library Architecture

The individual components of DarkRift are fairly self-explanatory. All code common to both server and client is held in the DarkRift namespace, all code for the server in DarkRift.Server and all code for the client in DarkRift.Client.

Underneath the both client and server namespaces are DarkRift.Client.Unity and DarkRift.Server.Unity which holds some Unity specific code to make it easier to use (e.g. the monobehaviours that we attached to the game objects). To use DarkRift you don’t need the code in these and it would be perfectly possible to build a Unity game using the standard interface in DarkRift.Client/DarkRift.Server but it makes it integrate a little better!

Pro users will also find a DarkRift.Server.Helpers namespace (in DarkRift.Server.Helper.dll) that contains a number of helper classes that can make writing server side plugins easier! We’ll look more into this in a moment.

Lastly there is a DarkRift.Logging namespace which is only used in the server but, since it is a sizable chunk of code and may be used in other packages in the future, is in its own library and namespace.

## Debugging

Before we get back into the practical aspects of DarkRift it is also very helpful to know how you can debug your plugins and applications.

Firstly, we discussed it earlier but the sniffer is a good way of telling you what data is getting to the server and what header data it contains. If you think something isn’t reaching the server then check with the sniffer!

Secondly you can use most .NET debuggers with DarkRift. If you copy the .pdb file that’s output with your plugin into the plugins directory with your plugins library, you can attach your IDE’s debugger to the DarkRift hosting process and then debug your code as you would any other code. Bear in mind however that Hazel (in its current state) does not play well with debuggers and connections may time out during paused execution!

Lastly, if all else fails, just print stuff to the console (see next chapter)!

## Logging

As we said above, there’s quite a bit of code to the logging system in order to facilitate it outputting to multiple sources and with different priorities. To plugins, however, there is a single method for logging: Plugin.WriteEvent.

WriteEvent takes a message, log type and an optional exception if it occurred and will handle the write out to the necessary source. The log type allows you to specify how important the event is and hence will also determine what files it is output to, dependent on the configuration file.

It’s recommended that you use WriteEvent fairly sparingly so that it’s easy to find the events later in your server logs but it is up to you how much you want to output!

## Managing plugins

There are 2 systems in DarkRift that oversee the plugins. The first is the PluginController which provides the interface commands for managing the plugins and the second is the PluginManager which handles the loading of plugins and generally performs lower level tasks. Whilst they have similar names they have very different purposes!

### PluginController

You may recall that we discussed how anything that takes command input is implemented internally in DarkRift as a plugin, a prime example of this is the PluginController.

Using the PluginController plugin you can uninstall any plugins previously loaded using

**plugins uninstall <plugin-name>**

And this will remove any information stored about them and any data they may have saved on the server.

You can also view the plugins installed and loaded (i.e. running) using

**plugins installed**

**plugins loaded (-h)**

Adding the **-h** flag will also display the internal plugins that are usually hidden.

### PluginManager

The PluginManager is a very different beast, however. When DarkRift starts up, the PluginManager is responsible for loading and keeping track of all plugins and hence invokes the install/upgrade process and keeps track of what version each plugin is expected to be.

We briefly saw earlier how we can access other plugins that are loaded in the server via the PluginManager using GetPluginByType but there are also methods GetPluginByName, GetInstalledVersion and GetAllPlugins plus an indexer that acts similarly to GetPluginByName. These can all be used to get information about other plugins and communicate between them as needed.

## Cheating

Since there are a few things DarkRift can detect that might indicate a hacked or malicious client (e.g. malformed packets, corrupt headers), DarkRift has a built-in system for tracking warnings that Pro users have access to.

Let’s head back to our Agar game and look at how we can add some

## Serialization?

## Persistent Storage

Leaderboard

## Commands and Interface

## Databases

## Installs and Upgrades

Change leaderboard table

## Dispatcher

# 5 Helpers

## Profanity Filter

## Rooms

# Appendix I: Upgrading from DarkRift 1