Appendix C. Combined Maze Runner Lessons

Table of Contents

Appendix C. Combined Maze Runner Lessons....... 1
C.0 Setting Up Our Workspace....... 4
C.1 Lesson #1 – Terrain for Our Game 8
C.2 Lesson #2 – loading datablocks....... 9
C.3 Lesson #3 – Game Coins....... 10
C.4 Lesson #4 – Fade and Fireball Blocks....... 12
C.5 Lesson #5 – Maze Blocks....... 16
C.6 Lesson #6 – Simplest Player....... 18
C.7 Lesson #7 – Preparing Our Game Inventory....... 23
C.8 Lesson #8 – Lava in the Cauldron....... 26
C.9 Lesson #9 – Starry Night....... 27
C.10 Lesson #10 – Low Lighting....... 28
C.11 Lesson #11 – Stormy Weather....... 29
C.12 Lesson #12 – Teleport Station Effect....... 31
C.13 Lesson #13 – Celestial Bodies....... 34
C.14 Lesson #14 – Teleport Stopper....... 35
C.15 Lesson #15 – Teleport Triggers....... 36
C.16 Lesson #16 – MoveMap....... 40
C.17 Lesson #17 – Level Loader....... 42
C.18 Lesson #18 – Game Events....... 50
C.19 Lesson #19 – FireBall Explosion....... 55
C.20 Lesson #20 – The FireBall....... 58
C.21 Lesson #21 – Game Sounds....... 61
C.22 Finishing the Prototype....... 66
C.23 Finish Gameplay Code....... 66
C.24 Improve Feedback....... 74

Note1 : The versions of the lessons included in this document may vary slightly from the book versions, but this will not affect the creation of your game.

C.0 Setting Up Our Workspace

Before we can work on any lessons, we must first set up a work area. Everything that you need to do this is supplied on the CD that comes with this guide. If you examine the CD you will find the following directories (and others):

- **“\Base”** - This directory contains data and scripts that are used in the lessons and can also be used later to make new games. Please see the “Lesson Kit Assets” appendix for additional information about the contents of this directory.
- **“\MazeRunner”** - Excluding the data and scripts in “\Base” and the executable from the demo kit, this directory contains all of the unique resources and scripts required to build the MazeRunner prototype.
- **“\MazeRunnerAdvanced”** - This directory contains a completed version of MazeRunner with several additional features as are suggested in Section 14.10 “Improving The Game”.
- **“\TorqueDemoInstallers”** - This directory contains installers for TGE.

At this time, if you do not have the demo installed on your machine, please do so by running the appropriate installer (based on your computer and operating system type). Once you have finished with that, please continue reading.

C.0.1 Starting from Torque Demo

First, let's make a new directory named "MazeRunner" and place it on a drive with at least 100 MB of free space. We'll want some elbow room while we work. In my case, the directory would be on my working drive here: "H:\MazeRunner".

Second, now that we have a place to work, let's copy the entire contents of the TGE demo directory into our new directory "MazeRunner".

C.0.2 Write Cleanup Scripts

It is a good idea to have the ability to remove temporary files from a working directory. If we remove all compiled scripts (DSOs) before re-running the engine, we are insuring that only new script content will be used. Additionally, it is a good idea to occasionally remove terrain lighting files (ML). To accomplish these two tasks, we will write some scripts.

The first script (if you are running Windows) will be called "DELDSO.bat". It is used to delete all compiled script files (DSO cleaning) and contains this simple line of script:

```
del /S /F *dso
```

In UNIX/Linux/OSX, the file would be "deldso", and the content of the file is:

```
rm -rf *dso
```

The second file (if you are running Windows) will be called "DELML.bat". It is used to delete all terrain lighting files (ML cleaning) and contains this simple line of script:

```
del /S /F *ml
```

In UNIX/Linux/OSX, the file would be "delml", and the content of the file is:

```
rm -rf *ml
```

We'll run the DSO cleaner each time we modify our scripts, and occasionally we'll run the ML cleaner to get rid of stale lighting files.

C.0.3 Copy MOD Directory

Although it is possible to modify the demo to create MazeRunner, it will be far simpler to start with a blank slate instead. To that end, a bare bones mod has been provided. To start with this mod, please copy "\MazeRunner\A_SettingUp\prototype" from the accompanying disk into "\MazeRunner".

C.0.4 Modify "main.cs"

Next, edit "main.cs" and change line 6 from this:

```
$defaultGame = "demo";
```

to this:

```
$defaultGame = "prototype";
```

This will use our new "prototype" mod instead of the demo mod.

C.0.5 Add Systems Scripts

The accompanying disk comes with a number of scripts that are provided to simplify your game-writing endeavors. We will be discussing some of these scripts in the guide, and those we do not discuss are documented in the "Scripted Systems" appendix.

From the accompanying disk, please copy the

"\Base\Scripts\EGSystems"

directory into

"\MazeRunner\prototype".

Then, edit the onStart() function in "\MazeRunner\prototype\main.cs" so it looks like this (BOLD lines are new code):

```
function onStart()
{
   // Maze Runner Changes Begin -->
   exec("./EGSystems/SimpleInventory/egs_SimpleInventory.cs");
   exec("./EGSystems/SimpleTaskMgr/egs_SimpleTaskMgr.cs");
   exec("./EGSystems/Utilities/egs_ArrayObject.cs");
   exec("./EGSystems/Utilities/egs_Misc.cs");
   exec("./EGSystems/Utilities/egs_Networking.cs");
   exec("./EGSystems/Utilities/egs_SimSet.cs");
   exec("./EGSystems/Utilities/egs_String.cs");
   // <-- Maze Runner Changes End

   //.. leave remaining code alone
```

C.0.6 Add Maze Runner Data

You are not expected to create your own content for this game. I have included all of the models, textures, and sounds you will need.

From the accompanying disk, please copy the

1. "\Base\Data\GPGTBase" directory into "\MazeRunner\prototype\data", and
2. "\MazeRunner\A_SettingUp\MazeRunner" directory into "\MazeRunner\prototype\data".

C.0.7 Create Maze Runner Scripts Directory

Although we will not be placing anything in yet, in preparation for our lessons, let's create the directory: "\MazeRunner\prototype\server\scripts\MazeRunner".

C.0.8 Test Run

After saving the modified "main.cs" and "prototype\main.cs", run the executable you placed in "MazeRunner" and the prototype should start up. If it does not, please retrace your steps and see if you missed something.

C.0.8.1. Windows users.

On Windows platforms, some users will get a warning about a missing or wrong sound setup. If, and only if, you get this message, copy the "\MazeRunner\A_SettingUp\OpenAL32.dll" file (found on the accompanying disk) into your "MazeRunner" directory and try again.

C.0.9 Ready To Start

OK, if you got the prototype to run, you're ready to start.

C.1 Lesson #1 – Terrain for Our Game

Alright, here is the first of several lessons in which we'll apply the massive amount of knowledge we're gaining in a practical situation, building our own simple game step-by-step. In this first quick lesson, we'll create a terrain for our game to be played out on. Follow the simple steps in this section to get started.

C.1.1. Copy required files.

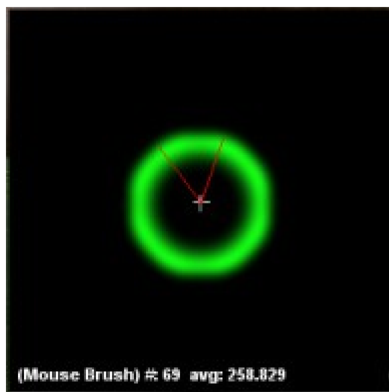
From the accompanying disk, please copy the "\MazeRunner\Lesson_001\heightFields"

directory into "\MazeRunner\prototype\data".

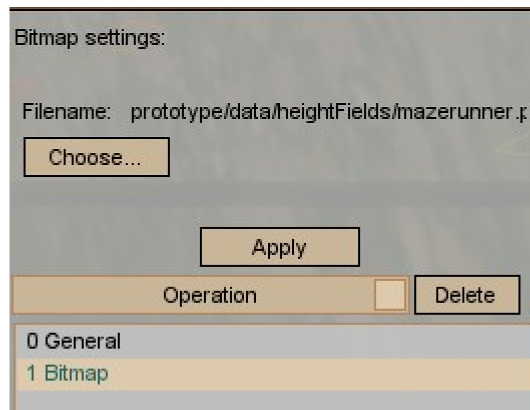
C.1.2. Generate new terrain

To generate the cauldron for our game terrain, do the following (see Figure C.1.1):

1. Start up your prototype.
2. Start the "Maze Runner" mission.
3. Start the Terraformer.
4. Use the 'Bitmap' operation to generate a terrain using the file "\MazeRunner\prototype\data\heightFields\mazerunner.png".



Terrain Preview



Terraformer Settings

Figure C.1.1. Terrain preview and Terraformer settings.

After applying the generator, the terrain should be shaped like a cauldron. Save the mission.

C.1.3. Adjust spawn point.

Now we have a simple terrain. You might also want to use the Inspector to move the spawn point to "0 0 100" so we don't have such a long ways to fall when we spawn into the mission again. Now, don't forget to save your changes.

And with that, we have taken the first small step towards making our little Maze Runner game!

C.2 Lesson #2 – loading datablocks

As we work on the Maze Runner Game, we are going to need several datablocks and the accompanying scripts that were created for your use in this game and in your future creations. So, let's take the time now to get them loading. From the accompanying disk, please

1. Copy the "\Base\Scripts\GPGTBase" directory into "\MazeRunner\prototype\server\scripts"

2. now, edit the function onServerCreated() in the file "\MazeRunner\prototype\server\game.cs" to look like this (BOLD lines are new or modified):

```
exec("./markers.cs");
exec("./player.cs");
exec("./GPGTBase/loadGPGTBaseClasses.cs"); // MazeRunner
```

In the above script, we are loading all of the Volume 1 base datablocks (classes) after all the other datablocks that FPS normally includes. We also added the data files that go with the datablocks.

To test for a successful load, simply start the prototype and load the "Maze Runner" mission. Then run the "Creator" tool and you should have directories in the creator as in Figure C.2.1.

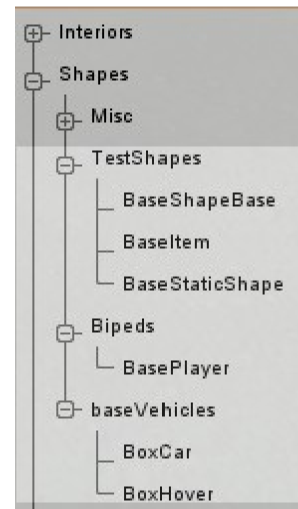


Figure C.2.1. Creator Directory

C.3 Lesson #3 – Game Coins

In this lesson, we will examine the game coin's datablock definition. Later, we will implement scripts to pick up these coins, but for now, all we need to do is talk about the coin's geometry, the datablock definition, and the creation script.

C.3.1. Copy required files.

From the accompanying disk, please copy the file

"\MazeRunner\Lesson_003\coins.cs"

into

"\MazeRunner\prototype\server\scripts\MazeRunner".

Now, edit the function onServerCreated() in the file: "\MazeRunner\prototype\server\game.cs" to look like this (BOLD lines are NEW or MODIFIED):

```
exec("./GPGTBase/loadGPGTBaseClasses.cs"); // MazeRunner
exec("./MazeRunner/coins.cs"); // MazeRunner
```

Please note, until this step, the "\MazeRunner\prototype\server\scripts\MazeRunner" directory did not exist, so you need to create it yourself.

C.3.2. Coin geometry.

The geometry for this coin is very simple, and can be found in: "\MazeRunner\prototype\data\MazeRunner\Shapes\Items\coin.ms3d", where we copied it earlier. If you load the file in Milkshape, you will see that it is nothing more than a thin disk. It has one render mesh and no collision mesh. Because this model is used for an item, a collision mesh will automatically be generated by TGE. The skin was generated using Ultimate Unwrap 3D. It's simple and does the job.

Now, all we need is a datablock and a creation script (onAdd()).

C.3.3. The coin datablock.

The datablock for our coins is very simple. If we look at the file we just copied, we will see this datablock definition:

```
datablock ItemData( Coin : BaseItem )
{
   shapeFile = "~/data/MazeRunner/Shapes/items/coin.dts";
   category  = "GameItems";
   sticky    = true;
   lightType = NoLight;
   mass      = 1.0;
   respawn   = false;
};
```

The coin item has the following attributes:

- It is an instance of Item (just to be clear about this).
- It is derived from BaseItem. (There are base datablocks for all of the classes we discuss in this guide.)

- We'll be able to find this object under "Shapes/GameItems" in the Creator menu.
- It is sticky and will stay put when it hits terrain or an interior.
- It does not emit light.
- As a rule, I never create a massless object. This avoids any future difficulties should I choose to apply an impulse to the shape. So, this coin gets an arbitrarily chosen mass of 1.
- When this coin is picked up we don't want it to be respawned. So, we set the field "respawn" to false. This won't mean anything to you yet, but when we discuss the Simple Inventory system in Chapter 7, "Gameplay Classes", this will become clear.

C.3.4. The coin onAdd().

We have mentioned callbacks only briefly thus far, and we will discuss them later in Chapter 9 "Gameplay Scripting". For now, just know that all SimObject instances and all instances of children of SimObject call the onAdd() callback after the object it created and initialized.

Later, when we write the scripts to place objects, it will become clear that we want objects to stay put when they are placed. Coins have the option of being static (won't move on their own), or 'non-static' (affected by gravity and other forces). Therefore, we need to force the coin to be static by making a suitable onAdd() callback. Find the following code at the end of the file we just copied:

```
function Coin::onAdd( %DB , %Obj )
{
   Parent::onAdd( %DB , %Obj );
   %Obj.static       = true;
   %Obj.rotate       = true;
}
```

The callback does the following:

- Calls the Parent:: version of this callback to allow it to do any work it needs to do (optional and based on your design methodology).
- Sets the object to a static object. Now, it won't fall (due to gravity) or be affected by impulses.
- Make the coin rotate. Now the render code will rotate the coin. Please note, this only rotates the render mesh, not the collision box that TGE generates.

C.3.5. Testing.

To verify that our changes worked, you can:

1. re-start the prototype,
2. open the "Maze Runner" mission,
3. start the Creator,
4. look under "Shapes" and find the folder "GameItems", and
5. open the "GameItems" folder to find a new placeable shape, "Coin".

If this did not work, check your console for errors (typos, files not found, etc).

C.4 Lesson #4 – Fade and Fireball Blocks

In our game, we are going to have two kinds of special maze blocks. The first one will be a block that can be faded in and out of view, and the second will be a block that shoots fireballs.

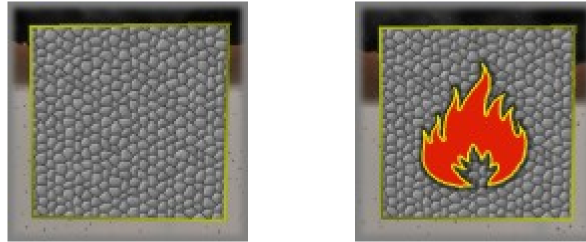


Figure C.4.1. Fade blocks (left) and Fireball blocks (right)

Both of these blocks require features from the ShapeBase hierarchy. The fade blocks use the fading and hiding features. The fireball block uses the reskinning property.

In this lesson, we will concentrate on the mesh properties and the datablocks that go with these two blocks. Later, we will write the scripts to fade the fade blocks, and to shoot fireballs from the fireball blocks.

C.4.1. Copy required files.

From the accompanying disk, please copy the:

1. file "\MazeRunner\Lesson_004\fadeblock.cs" into:

   "\MazeRunner\prototype\server\scripts\MazeRunner".

2. file "\MazeRunner\Lesson_004\fireballs.cs" into:

   "\MazeRunner\prototype\server\scripts\MazeRunner".

Then, modify onServerCreated() in "\MazeRunner\prototype\server\scripts\game.cs" to include these lines (BOLD lines are new):

```
exec("./MazeRunner/coins.cs"); // MazeRunner
exec("./MazeRunner/fadeblocks.cs"); // MazeRunner
exec("./MazeRunner/fireballs.cs"); // MazeRunner
```

C.4.2. Block geometry.

The blocks will both have the same geometry, namely a single render mesh and a single collision mesh. To see this geometry, open the file:

"\MazeRunner\prototype\data\MazeRunner\Shapes\MazeBlock\blockA.ms3d"

using Milkshape.

You will see that this model has a render mesh named "block0" and a single collision mesh named " collision-1".

To enable re-skinning, we need to do something special with the model's skin.

C.4.3. Re-skinning.

Still in MS3D, if you look at the material named "skin" you will see that we are using a texture named "base.skin.png". (It only shows as "base" on the MS3D button, but trust me, the file is named 'base.skin.png'. By using a skin with this name, we will later be able to change the skin on this model.

To clarify, the rules for reskinning are simple:

1. Skin your mesh with texture named "base.*XYZ*.png", where *XYZ* can be anything you choose. The important thing to notice is that the skin starts with "base.". This tells TGE this is a reskinnable mesh.
2. Create as many extra textures as you need, as long as they have the name "*LMN.XYZ*.png", where *XYZ* is the same name from step #1 and *LMN* is a name to make your texture name unique.
3. Reskin a shape at any time by writing this code:

```
%obj.setSkinName( "LMN");
```

The above code tells the mesh to use the texture "*LMN.XYZ*.png" instead of 'base.*XYZ*.png'.

C.4.3.1. Self-illuminating.

Because we are using a sort of cartoon/platform theme in our game, we will want all of the blocks to self-illuminate. This means that they will not be affected by the in-game lighting. To do this, I choose the self-illuminating option when exporting (using the DTS-Plus exporter). Please see Figure C.4.2.

Figure C.4.2. Making material self-illuminating

C.4.4. Datablocks.

Alright, these base blocks are pretty much good to go. Let's create some datablocks and we can move on.

C.4.4.1. Fade blocks datablock .

For the fade block, please open this file:

"\MazeRunner\prototype\server\scripts\MazeRunner\fadeblocks.cs".

In this file, and find the following lines of script:

```
datablock StaticShapeData( FadeBlock )
{
   category     = "FadeBlocks";

   shapeFile    = "~/data/MazeRunner/Shapes/MazeBlock/blockA.dts";

   isInvincible = true;
};
```

This datablock has the following attributes:

- Based on the value in 'category', these blocks will found in the Creator tree under "\Shapes\FadeBlock".
- It loads the mesh for the model we just discussed.
- It is invincible and thus takes no damage. We want this so that fireballs striking a fadeblock will not damage it.

Not shown, but present in the completed copy of this file (as written by me), there is another bit of code at the top. It is a reloader. Reloaders are little scripts that are used to re-load the file, thus re-loading the datablock definitions and any scripts in the file. In single-player mode, I use reloaders to reload files I have changed while the mission is still running. This way I can make minor tweaks to scripts, etc. and not have to reload the entire mission.

The reloader for the "fadeblocks.cs" file would be:

```
function rldfb()
{
   exec("./fadeblocks.cs");
}
```

C.4.5. Fireball blocks datablock .

For the fade block, please open this file:

"'\MazeRunner\prototype\server\scripts\MazeRunner\fireball.cs"

In this file, and find the following lines of script:

```
datablock StaticShapeData( FireBallBlock )
{
   category     = "FireBallBlocks";
   shapeFile    = "~/data/MazeRunner/Shapes/MazeBlock/blockA.dts";
   isInvincible = true;
};
```

As you can see, this datablock is identical (except for the name) to our fadeblock datablock. The behavior differences are entirely script based, and the only reason we need another datablock is for scoping. We'll get to this later.

C.5 Lesson #5 – Maze Blocks

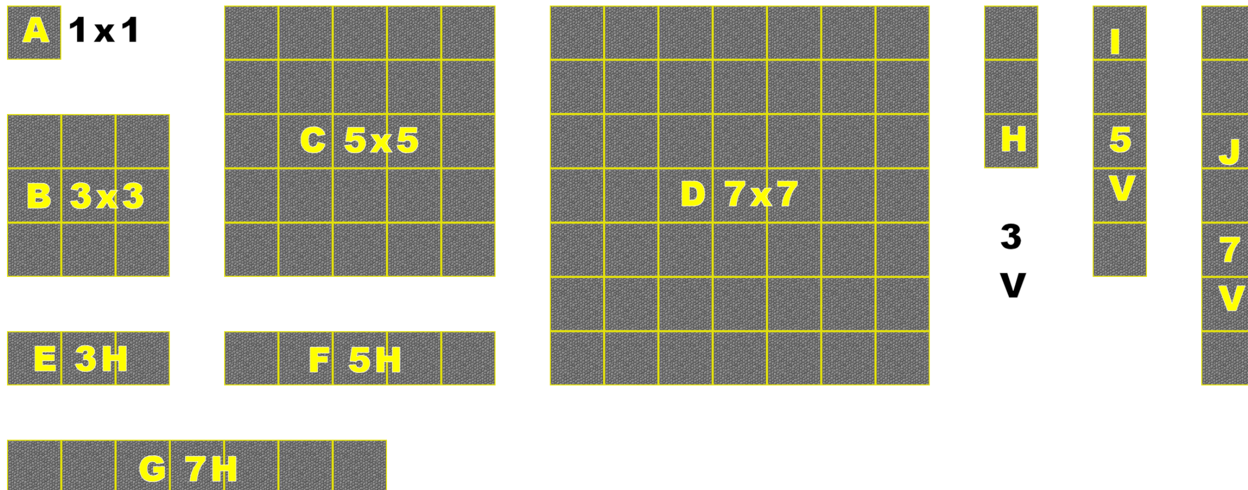
The primary geometry of our maze consists of blocks and groups of blocks. Later, when we discuss the level building scripts, we'll talk about how these blocks are placed. For now, we will restrict ourselves to the creation of these blocks.

The MazeBlocks share the same geometry and skin set up as the fade blocks and fireball blocks from Lesson #4 (above). So, if you have not completed that lesson please do it first.

C.5.1. Block Geometry

In addition to the single block geometry we produced for the prior blocks, we need several additional variations for the maze blocks. In theory, we could build our entire level out of single blocks. However, I don't advise this as we do pay a penalty (network and processing) for each block in the scene. So, knowing in advance that we will have various structures in our levels combining several blocks, we will make a few larger meshes. This way if we need an area the size of say nine (3x3) blocks, we can place just one big block.

If you look in the ''\MazeRunner\prototype\data\MazeRunner\Shapes\MazeBlock' directory we created earlier, you will see that there are blocks A through J. Those shapes have the following geometries (as seen from the top down):



As you may notice, there are four square blocks, and three each of the 'linear' block for horizontally oriented and vertically oriented. It may not be apparent immediately, but with these blocks we can create symmetrically laid out levels without needing to re-orient the blocks at placement time.

C.5.2. Placing Blocks

We aren't writing the code to place these block yet, but when we do it will look something like this:

```
new TSStatic()
{
   shapeName = "~/data/MazeRunner/Shapes/MazeBlock/block" @ %blockType @ ".dts";

   position  = %actX SPC %actY SPC $CurrentElevation;

   scale     = "1 1 1";
};
```

This code snippet is actually from our level builder and as you can see, we will be dynamically selecting the mesh to use as well as calculating the position as we place the block.

C.5.2. Examine The Blocks

This guide does not discuss modeling, nor does it cover the various modeling tools. However, as the blocks have already been created for you, I suggest that you examine a few to see how they are constructed. Pay particular attention to blocks E through J.

Don't forget, all of the mazeblocks have been copied over to our data directory already at:

"\MazeRunner\prototype\data\MazeRunner\Shapes\MazeBlock".

You can open any of the block models (*.MS3D) with a copy of Milkshape.

C.6 Lesson #6 – Simplest Player

For our game, we will need to make a very simple player. This player is nothing more than a ball with three nodes (joints), root, eye, and cam (Figure C.6.1).

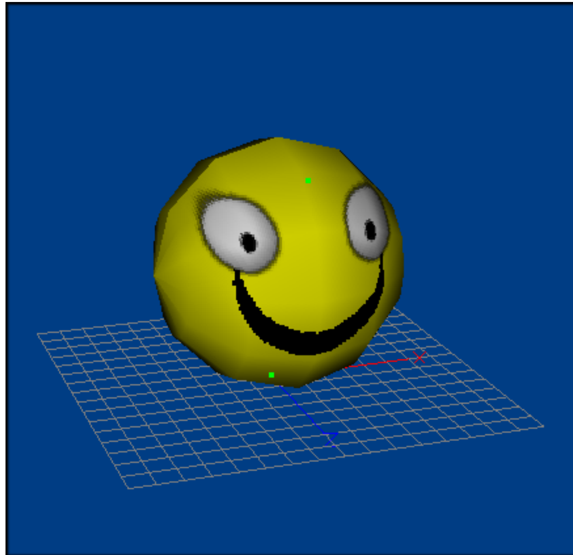


Figure C.6.1. Simplest Player.

C.6.1. Copy required files.

From the accompanying disk, please copy the file "\MazeRunner\Lesson_006\mazerunnerplayer.cs" into "\MazeRunner\prototype\server\scripts\MazeRunner".

Now, edit the function onServerCreated() in the file "\MazeRunner\prototype\server\game.cs" to look like this (BOLD lines are new or modified):

```
exec("./MazeRunner/fireballs.cs"); // MazeRunner
exec("./MazeRunner/mazerunnerplayer.cs"); // MazeRunner
```

C.6.2. Simplest Player skeleton.

Because we're not going to animate this player, it doesn't need very many nodes (joints) in its skeleton. In fact it only needs a root node and the two camera mount points (see Table C.6.1).

Table C.6.1. Simples Player nodes.

Node	Description
root	The root node, specifying the physical bottom of the mesh.
eye	The 1st POV camera mount.
cam	The 3rd POV camera mount.

C.6.2.1. Root node.

In this model, the root node is located at the bottom of the player and all vertices in the player are assigned to it. This node defines the bottom of the player and is where mesh contacts the ground. If this node were placed in the middle of the player, the player would sink into the ground.

C.6.2.2. Eye and Cam Nodes

The next node is the eye node. It is located on the "forehead" just above and between the eyes. This is where the 1st POV camera will be mounted.

The last node is the cam node. This is located behind and above the model. It doesn't necessarily need to be here, but this model was designed (in part) to show the difference between an eye mount and a cam mount. As you've probably guessed, this is where the 3rd POV camera will mount.

C.6.3. Simplest Player geometry.

C.6.3.1. Visible mesh.

There isn’t much to say about this. It's a ball. The player has one mesh and one skin. We're not using any IFLs or other fancy features.

C.6.3.2. Collision mesh.

We do not need to define a collision mesh for instances of the Player class, as the engine does this automatically.

C.6.4. Simplest Player animations.

Earlier I said that this player is not animated. I lied. OK, I didn't exactly lie. For any player to work, the root animation needs to be exported at a minimum. Then, to get rid of some annoying warnings, you'll need to export the other animations (shown in Table C.6.2).

Since the player isn't going to need these animations, I've left them blank and just exported the same sequence for each.

Table C.6.2. Animation descriptions.

Animation	Description
root	A default animation which plays while the player is at rest.
run	Forward running animation.
back	Backwards running animation.
side	Sideways stepping animation.
jump	Moving jump animation.
standjump	Stationary jump animation.
fall	Long falling animation which starts about 1 second after fall starts.
land	Hard landing animation. (Played while in recovery-mode).

The sequences for these animations are shown in Table C.6.3.

Table C.6.3. Sequences for animations.

Animation	Start Key	End Key	FPS	Cyclic	Blended
root	1	2	1	Y	N
	seq: root=1-2, fps=1, cyclic				
run	1	2	1	Y	N
	seq: run=1-2, fps=1, cyclic				
back	1	2	1	Y	N
	seq: back=1-2, fps=1, cyclic				
side	1	2	1	Y	N
	seq: side=1-2, fps=1, cyclic				
jump	1	2	1	N	N
	seq: jump=1-2, fps=1, cyclic				
standjump	1	2	1	N	N
	seq: standjump=1-2, fps=1				
fall	1	2	1	N	N
	seq: fall=1-2, fps=1, cyclic				
land	1	2	1	N	N
	seq: land=1-2, fps=1, cyclic				

Table C.6.3. shows the following information:

- **Animation** – This is the (required) name for the animation sequence in question.
- **Start Key/End Key** – These are the frames in which the named animation begins and ends.
- **FPS** – This is the base frame-rate at which the animation should be played.
- **Cyclic** – This indicates whether the animation should be played once or in a cycle.
- **Blended** – This indicates whether the sequence should be blended or not.

Finally, for each sequence there is a combined line something like "seq: root=1-2, fps=1, cyclic." This is what you would type in for the default exporter, but since we're using the DTS Plus exporter, you will enter the values via that exporter's dialog. See the "MS3D Cheat Sheets" in the appendix for a quickie tutorial on exporting animations with DTS Plus. Also note, the old exporter does not support blending.

C.6.5. Simplest Player's datablock.

Because the datablock for this shape is a bit long, only the pertinent portions are listed here:

```
datablock PlayerData( MazeRunner : BasePlayer )
{
   shapeFile            = "~/data/MazeRunner/Shapes/Players/MazeRunner.dts";

   boundingBox          = "1.6 1.6 2.3";

   invincible           = true;

   groundImpactMinSpeed = 1000;
   ImpactMinSpeed       = 1000;

   renderFirstPerson    = false;

   observeThroughObject = true;

   // ...
};
```

This player has the following notable attributes:

1. It derives (copies) from the BasePlayer datablock that comes on the accompanying disk.
2. As would be expected, the mesh we just built (or copied) is used.
3. The shape is a little bigger than the normal character, so we've increased the dimensions of its bounding box from "1.2 1.2 2.3" to "1.6 1.6 2.3," adding an extra three-tenths of meter in the x and y dimensions.
4. The player is marked as invincible because we are not going to use damage to determine if it is "Dead." Instead, we'll kill it immediately if the mesh is hit by a fireball or if it falls in the lava.
5. Impacts are effectively disabled by setting the velocities above any velocity the player will be able to achieve in this game.
6. renderFirstPerson is disabled, meaning the mesh will not render in 1st POV.
7. The camera has been instructed to use the player's camera settings (observeThroughObject is true).

To get the entire datablock, please copy: "MazeRunnerPlayer.cs" (**MazeRunner_008**) to "\MazeRunner\prototype\server\scripts\MazeRunner"

C.6.6. Loading the datablock.

Now, edit the "\MazeRunner\prototype\server\scripts\game.cs" file and update onServerCreated() to contain this code (BOLD is new code):

```
exec("./MazeRunner/fireball.cs"); // MazeRunner
exec("./MazeRunner/MazeRunnerPlayer.cs"); // MazeRunner
```

C.6.7. Using this player.

Now, to use this player instead of the Blue Guy we have been using thus far, edit the "\MazeRunner\prototype\server\scripts\game.cs" file and modify the highlighted code (below) in GameConnection::createPlayer() to look like this:

```
function GameConnection::createPlayer(%this, %spawnPoint)
{

   //...

   // Create the player object
   %player = new Player()
   {
      dataBlock = MazeRunner; // Change this line
      client = %this;
   };

   //...
```

C.7 Lesson #7 – Preparing Our Game Inventory

In this short lesson, we will examine the steps required to get our player (MazeRunnerPlayer) to use the Simple Inventory system to pick up coins.

C.7.1. Loading the inventory system.

In order to use our inventory system, we must ensure that it is getting loaded. In fact, we have already done this first step. When we set up our "MazeRunner" directory and prepared a copy of prototype mode, we modified the file "\MazeRunner\prototype\main.cs". We had it load the inventory system's main script file:

```
function onStart() // in main.cs
{
  exec("./EGSystems/SimpleInventory/egs_SimpleInventory.cs"); // MazeRunner
  exec("./EGSystems/SimpleTaskMgr/egs_SimpleTaskMgr.cs"); // MazeRunner

   //..
```

This then loaded the other script files that compose this system.

```
// in egs_SimpleInventory.cs

exec("./SimpleInventoryBuilder.cs");
exec("./SimpleInventoryGeneral.cs");
exec("./SimpleInventoryValidation.cs");
```

C.7.2. Adding an inventory.

With the inventory system being loaded, we now have to hook it to any classes that wish to "own" an inventory. The simplest way to do this is to have each class add an inventory system to the object when the object's onAdd() callback is executed.

Take a look in this file:

"\MazeRunner\prototype\server\scripts\GPGTBase\Player\PlayerDataConsoleMethods.cs"

It contains the definitions for all of the important callbacks used by a player class. All of these callbacks are scoped to PlayerData::, ensuring that they will be called unless a new datablock, deriving from PlayerData::, redefines the callbacks.

We are already loading this script file, so we get the benefit of all of these callbacks already. One of these callbacks is PlayerData::onAdd(), which, among the other things that it does, creates an inventory and saves a reference to it in the player object.

```
function PlayerData::onAdd(%DB,%Obj)
{
   // 1
   Parent::onAdd(%DB,%Obj);

   // 2
   %Obj.enableMountVehicle = true;

   // 3.
   %Obj.myInventory = newSimpleInventory();

   %Obj.myInventory.setOwner(%Obj);
}
```

This means, we do not have any work to do. We do not have to implement a new version of onAdd() scoped to MazeRunnerPlayer::, but if we wanted to, we could write one like this:

```
function MazeRunner::onAdd( %DB , %Obj )
{
   Parent::onAdd( %DB , %Obj ); // Usually called firstPersonOnly

   // Other statements here ...
}
```

C.7.3. Removing an inventory.

It is normal to destroy objects created in the onAdd() callback, when the onRemove() callback is executed.

Again, this is taken care of for us by the base code we are using from the prototype. Here is the onRemove() callback from the same file we just examined above:

```
function PlayerData::onRemove(%DB,%Obj)
{
   // 1
   if( isObject( %Obj.myInventory )  )
      %Obj.myInventory.delete();

   // 2
   Parent::onRemove(%DB,%Obj);
}
```

Easy as pie! Of course, we could again write a specialized version of the onRemove() callback and just be sure to call the Parent:: version at some point (normally last):

```
function MazeRunner::onRemove( %DB , %Obj )
{
   // Other statements here ...

   Parent::onRemove( %DB , %Obj ); // Usually called last
}
```

C.7.4. What about constraining?

In our game, we don't want to constrain the inventory, but if we wanted, for some reason, to prevent the player from picking up coins, we could simply modify the onAdd() callback to look like this:

```
function MazeRunner::onAdd( %DB , %Obj )
{
   Parent::onAdd( %DB , %Obj );

   %obj.myInventory.setInventoryMaxCount( Coin ,  0 ); // No coins for you!
}
```

C.7.5. In review.

I know you're disappointed that there was no work to do in this lesson. So, let's just summarize the steps instead. This way you will know what they are when you are on your own.

1. Load inventory system scripts.
2. Ensure that the onAdd() callback adds an inventory to the object when it is created.
3. In your own onAdd(), be sure to constrain the inventory system as is required by your game. Use the constraint methods included with the inventory system.
4. Make sure that the onRemove callback deletes the inventory.

C.8 Lesson #8 – Lava in the Cauldron

The game will have lava at the bottom of the cauldron. Falling into this lava kills the avatar and causes it to be respawned in its original spawn position. For now, we're only worried about getting the visual part done (the lava). We'll handle the interactions later.

For now, please do the following:

1. Start up your prototype, run the "Maze Runner" mission, and start the Creator Tool.

2. Create a water block (Mission Objects -> Environment -> Water), only providing the name "MazeRunnerWater" when the creator dialog appears (Figure C.8.1).

Figure C.8.1. Creating a water block.

3. Using the Inspector, be sure that the water has the settings shown in Table C.8.1.

Table C.8.1. Water settings for lava.

Parameter	Value
position	-256 -256 55
scale	512 512 15
UseDepthMask	true
surfaceTexture	prototype/ data/GPGTBase/water/lava.png
shoreTexture	prototype/ data/GPGTBase/water/lava.png
specularMaskTex	prototype/ data/GPGTBase/water/lavaspecmask.png
specularColor	1 1 1 0.2
specularPower	12
All others	Use defaults

OK, so it doesn't look exactly like lava, but it gets the point across. You can tweak this to your heart's content after we get the game running. For now, let's move on.

C.9 Lesson #9 – Starry Night

If you are building the Maze Runner game while you read this guide, the original sky is a bit too bright for our game, so we will need to do the following to create a starry night instead.

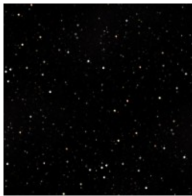
1. Start up your prototype, run the "Maze Runner" mission, and start the Inspector Tool.
2. Find the Sky object and change the DML file to one you will find in

   "/MazeRunner/prototype/data/GPGTBase/skies/starrynight/starry_sky.dml".

This file contains this list of texture names:

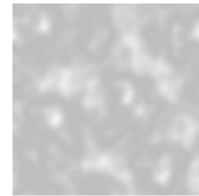
```
stars0
stars1
stars2
stars3
stars4
stars5
stars6
cloud1
```

The textures used in this file are just a set of five generated starfields, a placeholder for the seventh texture, and a randomly (noise) generated translucent cloud texture (Figure C.9.1).



sky0 .. sky5 (similar)

sky6 (placeholder)



cloud1

Figure C.9.1. Sky textures.

3. Using the Inspector, be sure that the water has the settings show in Table C.9.1.

Table C.9.1. Sky settings for starry night.

Parameter	Value
materialList	prototype/ data/GPGTBase/skies/starrynight/starry_sky.dml
cloudHeightPer[0]	0.5
cloudHeightPer[1]	0
cloudHeightPer[2]	0
cloudSpeed1	0.0005
cloudSpeed2	0
cloudSpeed3	0
visibleDistance	1000
fogDistance	2000
fogVolume1	550 0 300
fogVolume2	0 0 0
fogVolume3	0 0 0
all others	Use defaults

C.10 Lesson #10 – Low Lighting

If you are building the Maze Runner game while you read this guide, the original sun (lighting) is a bit too bright for our game, so we will need to do the following to match our night sky.

1. Using the Inspector, lower the lighting values for the Sun object to the values in Table C.10.1.

Table C.10.1. Lighting values for low lighting.

Fields	Values
elevation	90
azimuth	90
color	0.5 0.3 0.3 1
ambient	0.2 0.2 0.2 1

Now, relight the scene (ALT + L) to see the values take effect.

C.11 Lesson #11 – Stormy Weather

If you are building the Maze Runner game while you read this guide, we are now going to add some rain, lightning, and thunder to our scene. The game is meant to have a "cartoon spooky" theme, and these element will add to that.

C.11.1. Adding the rain.

1. Start up your prototype, run the "Maze Runner" mission and start the Creator tool.

2. Select a precipitation object (Mission Objects -> Environment -> Precipitation), giving it the object name "MazeRunnerRain" and choosing the datablock BaseRain (Figure C.11.1).

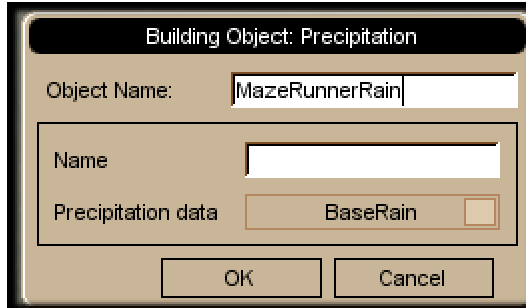


Figure C.11.1. Adding rain.

3. Open the Inspector and give the new rain object the settings in Table C.11.1.

Table C.11.1. Settings for rain.

Parameter	Value
minSpeed	1
maxSpeed	1.5
rotateWithCameraVel	true
numDrops	2000
boxWidth	200
boxHeight	100
doCollision	0
all others	Use defaults

C.11.2. Adding the lightning and thunder.

1. Go back into the Creator tool.

2. Select a lightning object (Mission Objects -> Environment -> Lightning), giving it the object name "MazeRunnerLightning" and choosing the datablock BaseLightning (Figure C.11.2).

Figure C.11.2. Adding lightning.

3. Open the Inspector and give the new lightning object the settings in Table C.11.2.

Table C.11.2. Settings for lightning.

Parameter	Value
position	0 0 300
scale	256 256 250
strikesPerMinute	6
strikeWidth	1.5
strikeRadius	128
color	0.89 0.8 0.42 1
fadeColor	0.5 0.9 0.9 1
chanceToHitTarget	0
boltStartRadius	32
all others	Use defaults

C.12 Lesson #12 – Teleport Station Effect

If you are building the Maze Runner game while you read this guide, we are now going to create the datablocks for a set of particle emitters that will be used later to mark the position of our teleport stations. We will need three distinct versions of this emitter. So, our strategy will be to create a base ParticleData datablock and a base ParticleEmitterData datablock using the previous ParticleData datablock. Then, we will use the inheritance feature of TorqueScript to create two copies of each datablock with minor modifications. This will give us a total of six datablocks. For the ParticleEmitterNodeData datablock, we'll just use the basePEND datablock that comes with this guide.

C.12.1. Copy required files.

From the accompanying disk, please copy the file

"\MazeRunner\Lesson_012\teleporters.cs"

into

"\MazeRunner\prototype\server\scripts\MazeRunner".

Now, edit the function onServerCreated() in the file "\MazeRunner\prototype\server\game.cs" to look like this (BOLD lines are new or modified):

```
   exec("./MazeRunner/mazerunnerplayer.cs"); // MazeRunner
   exec("./MazeRunner/teleporters.cs"); // MazeRunner
```

C.12.1.1. ParticleData (TeleportStation_PD0).

We want our particles to be nebulous particles of medium size with a red, green, or blue coloration.

```
datablock ParticleData(TeleportStation_PD0)
{
   dragCoefficient      = 0.0;
   gravityCoefficient   = -0.50;
   inheritedVelFactor   = 0.0;
   constantAcceleration = 0.0;
   lifetimeMS           = 400;
   lifetimeVarianceMS   = 100;
   useInvAlpha          = false;
   textureName          = "~/data/GPGTBase/particletextures/smoke";
   colors[0]            = "0.7 0.1 0.1 0.8";
   colors[1]            = "0.7 0.1 0.1 0.4";
   colors[2]            = "0.7 0.1 0.1 0.0";
   sizes[0]             = 0.1;
   sizes[1]             = 0.3;
   sizes[2]             = 0.3;
   times[0]             = 0.0;
   times[1]             = 0.5;
   times[2]             = 1.0;
};
```

As can be seen,

- this particle will float upward since it has a negative gravity coefficient;
- it has a short lifetime between 300 and 500 milliseconds;
- the particle it uses is nebulous (see negative image in Figure C.12.1);
- it fades from medium red to dark red evenly; and
- it starts off small and triples in size over time.

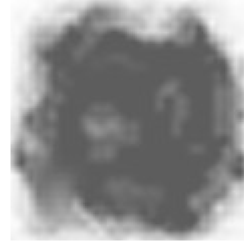


Figure C.12.1. Smoke particle.

C.12.1.2. ParticleEmmitterData (TeleportStation_PED0).

```
datablock ParticleEmitterData(TeleportStation_PED0)
{
   ejectionPeriodMS = 1;
   periodVarianceMS = 0;
   ejectionVelocity = 2.0;
   ejectionOffset   = 0.5;
   velocityVariance = 0.5;
   thetaMin         = 0;
   thetaMax         = 80;
   phiReferenceVel  = 0;
   phiVariance      = 360;
   overrideAdvance  = false;
   particles        = "TeleportStation_PD0";
};
```

As can be seen,

- This particle emitter ejects a new particle every millisecond, meaning we'll have up to 500 particles alive at any time (per emitter);
- it ejects particles at 1.5 to 2.5 meters per second starting at the center to 0.5 meters out;
- the ejection vector will be anywhere about the center and starts from slightly upward to straight up; and
- of course, it uses the particle we just made.

C.12.1.3. Duplicate datablocks.

The last step before trying these emitters out is to duplicate them so we have three sets. As you can see when looking at the code, we have taken advantage of TGE's datablock inheritance:

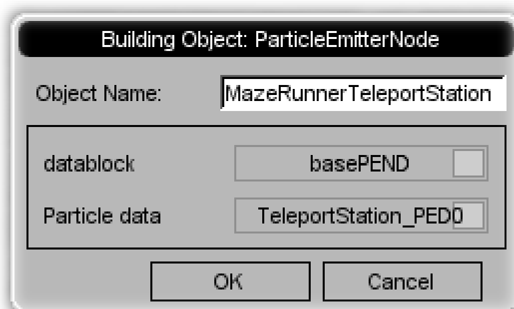
```
datablock ParticleData(TeleportStation_PD1 : TeleportStation_PD0 )
{
   colors[0]     = "0.1 0.7 0.1 0.8";
   colors[1]     = "0.1 0.7 0.1 0.4";
   colors[2]     = "0.1 0.7 0.1 0.0";
};

datablock ParticleEmitterData(TeleportStation_PED1 : TeleportStation_PED0 )
{
   particles        = "TeleportStation_PD1";
};

datablock ParticleData(TeleportStation_PD2 : TeleportStation_PD0 )
{
   colors[0]     = "0.1 0.1 0.7 0.8";
   colors[1]     = "0.1 0.1 0.7 0.4";
   colors[2]     = "0.1 0.1 0.7 0.0";
};

datablock ParticleEmitterData(TeleportStation_PED2 : TeleportStation_PED0 )
{
   particles        = "TeleportStation_PD2";
};
```

We only needed to change the particle colors and to use the correct particle in our new emitters.

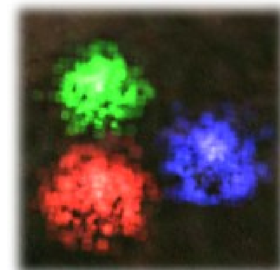
C.12.1.4. Testing the emitters.

We're not ready to use these emitters in our game, but we should test them. Do the following:

1. Restart the prototype.
2. Load the "Maze Runner" mission.
3. Use the Creator to place a particle emitter ("Mission Objects -> Environment -> ParticleEmitter").
4. Give the emitter (node) any name you like.
5. Use the basePEND ParticleEmitterNodeData datablock.
6. Select one of the three ParticleEmitterData datablocks we just examined (Figure C.12.2).



ParticleEmitter Dialog Settings



Resultant Emitters

Figure C.12.2. Testing the emitters.

C.13 Lesson #13 – Celestial Bodies

If you are building the Maze Runner game while you read this guide, we are now going to create some celestial bodies to go with our game. I have to apologize, but the celestial bodies we will implement are just too darn big (in terms of code) to show in the book. Instead, I will summarize their behaviors here and allow you to look at the scripts yourself.

C.13.1. Loading the celestial bodies.

The celestial bodies example as been created for you. To add it to the Maze Runner mission, follow these steps:

1. Open the file "/MazeRunner/prototype/data/missions/mazerunner.mis"
2. Open the file "\MazeRunner\Lesson_013\CelestialBodies.cs" and copy the contents into your copy-buffer (like you are doing a copy-paste operation).
3. Paste the data you just copied into the "mazerunner.mis" file before these lines:

```
};
//--- OBJECT WRITE END ---
```

Now, you can restart the prototype, load the Maze Runner mission, and you should see three celestial bodies in the sky.

C.13.2. Dying star.

The first celestial body is the "Dying Star". This celestial body is designed to represent a sun in our game world solar system. This sun is approaching the end of it's life and has shifted from yellow to red. To create the effect of a sun with moving sun spots, I have animated the brightness, the coloration, the size, and the rotation. Together with the image we are using for the sun, it may give the illusion of an active sun surface.

C.13.3. Far planet.

The next celestial body is the "Far Planet". This celestial body is designed to represent a distant planet in our game world solar system. It is stationary relative to the planet we are on.

C.13.4. Near moon.

The last celestial body is the "Near Moon". This celestial body is designed to represent a moon rotating about our planet. Its azimuth changes slowly over time; during this transition it rises and falls in the sky.

C.14 Lesson #14 – Teleport Stopper

When the player runs into a teleport station, we'd like the avatar to be stopped. To do this, we can use a P-zone set up as follows:

```
%Pzone = new PhysicalZone()
{
     position     = vectorAdd( "1 -1 0" , %Obj.getPosition() );
     rotation     = "1 0 0 0";
     scale        = "2 2 4";
     velocityMod  = "0";
     gravityMod   = "1";
     appliedForce = "0 0 0";
     polyhedron   = "0 0 0 1 0 0 0 -1 0 0 0 1";
};
```

As can be seen, this code is meant to be script driven. That is, we'll be substituting values in for position when we drop the object into the world.

The key things to notice are:

1. The position is offset by a vector (we haven't discussed vectorAdd() yet, but it adds two vectors and returns the result). The reason for this is that the polygon used to define the polyhedron field is offset. Its corner is at the origin and therefore the cube is not centered. This can either be corrected either by changing the polyhedron values or by offsetting while placing. I chose the latter.
2. velocityMod is set to zero. This means that shapes entering the P-zone should stop moving.

That is pretty much all for now. Later, we'll use this code in our teleporter building scripts.

C.15 Lesson #15 – Teleport Triggers

In this lesson, we will examine the scripts needed to teleport the player from one teleport station to another. We will also look at the code that combines the prior parts of the teleport trigger components.

C.15.1. Trigger datablocks.

We could in theory use the DefaultTrigger datablock that comes with the prototype, but it would be better to define a new one so we can guarantee that we have a unique namespace to scope our methods and callbacks with. So, we will define our datablock like this:

```
datablock TriggerData(TeleportTrigger)
{
    tickPeriodMS = 100;
};
```

C.15.2. Teleport scripts.

Later, when we are writing our level-building scripts, it will be nice if we already have a method for attaching the particle effects and the physical zone to our teleporter triggers.

With a little planning, this won't be that hard to do.

C.15.2.1. Teleport trigger planning.

We haven't discussed it much yet, but the user (and we) will be able to define new levels by creating simple text files. These files will have maps of the level in them made up of various numbers and letters, representing the positions of level pieces like blocks, coins, and teleporters.

Knowing that our teleporters will be associated with maker letters in this file, we can plan on the teleporter trigger being a sort of parent. We will read the level file, create a trigger where it tells us to, and store information in the trigger that tells the trigger which of the three types of teleporters it is. Recall there are three types of teleporters (all of them function the same, but this allows us to have distinct sets that are connected to each other).

So, let's just assume that the letters used to represent teleporters are going to be x, y, and z. Furthermore, let's assume that the trigger is created first and then the type is stored a field named "type." Lastly, we will assume the the level loader will then call our teleport builder script to add a particle emitter node and a P-zone in the same position as the trigger.

C.15.2.2. Teleport trigger implementation.

All of that planning and assuming gets us some code like this:

```
function Trigger::AttachEffect( %Obj )
{

   echo("\c5 Added Teleport Trigger");

   %emitter[X] = "TeleportStation_PED0";
   %emitter[Y] = "TeleportStation_PED1";
   %emitter[Z] = "TeleportStation_PED2";

   %effect = new ParticleEmitterNode() {
      position  = vectorSub(%Obj.getWorldBoxCenter(), "0 0 2");
      rotation  = "1 0 0 0";
      scale     = "1 1 1";
      dataBlock = "basePEND";
      emitter   = %emitter[%Obj.type];
      //emitter   = "TeleportStation_PED0";
      velocity  = "1";
   };

   %Obj.myEffect = %effect;

   %Pzone = new PhysicalZone() {
      position     = vectorAdd( "1 -1 0" , %Obj.getPosition() );
      rotation     = "1 0 0 0";
      scale        = "2 2 4";
      velocityMod  = "0";
      gravityMod   = "1";
      appliedForce = "0 0 0";
      polyhedron   = "0.0000000 0.0000000 0.0000000 1.0000000 0.0000000 0.0000000
0.0000000 -1.0000000 0.0000000 0.0000000 0.0000000 1.0000000";
   };
   %Obj.myPzone = %Pzone;

}
```

Basically, this function creates an array of particle emitter datablock names indexed by x, y, and z. Then, it creates a particle emitter in the position of the trigger (ID passed as argument to this function) and looks at the stored type to de-reference the datablock array, getting the correct datablock name to match the type.

Next, the function creates a P-zone (remembering to offset it a little) in the same position as the trigger.

After this function is finished executing there will be a trigger, a particle emitter node, and a P-zone all in the same location. Viola! A teleporter station.

So, how do we make the teleporter go? Let's do that next.

C.15.3. Trigger callbacks.

To make the teleporters do work for us, we need to implement the onEnterTrigger() and onLeaveTrigger() callbacks.

Instead of showing you the code (which you can just load and examine), I will present the methodology used to teleport correctly.

C.15.3.1. onEnterTrigger().

This callback has the lion's share of the work. Initially, all triggers will start off active. When the avatar runs into a teleporter trigger, that trigger executes the following steps:

1. Check to see if it is disabled. If so, the callback aborts.
2. Enable the P-zone it owns (to be sure the avatar gets stopped).
3. Check for the existence of other triggers. When we create triggers in our level building scripts, all triggers are added to one of three trigger groups based on their type. Then, if we recall that all SimObject children can determine what group (if any) they are stored in, it will become clear that each trigger can get the group it belongs in and choose a trigger from that group until finds one that is not itself.
4. Disable the P-zone on the target trigger (so player is not stopped on exiting that trigger).
5. Disable the target trigger (to prevent teleport loops).
6. Do some fading effects and schedule a setTransform() call, moving the avatar to the location of the target trigger.

Once the avatar arrives at the target trigger, it has to leave that trigger to reactivate it. Also note, the setTransform() call moves the avatar and causes the current trigger to call its own onLeaveTrigger() callback, thus reactivating it.

C.15.3.2. onLeaveTrigger().

This callback has very little to do. Basically, when the avatar leaves the trigger area, the trigger will be told to re-enable the trigger and to reactivate() the P-zone.

C.15.3.3. Tricky bits.

While examining the scripts, you may notice a couple of bits of code that we have not yet discussed. The first of these is a call to getRandomObject(). It is being called on a SimGroup. This is a method I have provided in the included systems script files (loaded when we set up our environment). This method simply iterates over a SimSet (or child) and randomly selects an entry from the set, returning the ID of the selection.

```
... %Trigger.getGroup().getRandomObject()  ...
```

The second bit involves the use of the function getWords(). In this line of code, we're replacing the position part of the player's transform with a new position while retaining the player's orientation information. This is done by getting the words representing the orientation. As you will learn later, a word is any string, and words are separated by spaces. Thus, we can look at the transform as a string containing seven words. Using getWords(), we simply get the top four words and then we paste them onto a new position matrix, making a new transform:

```
%newTransform = %newPos SPC getWords( %oldTransform, 3 , 6);
```

C.15.4. Trigger cleanup.

It is also worth mentioning that when the trigger is destroyed, it will call its onRemove() callback , which will delete the effects attached to this trigger. Nice and clean.

```
function TeleportTrigger::onRemove( %DB, %Obj )
{
   if( isObject( %Obj.myEffect ) )
      %Obj.myEffect.delete();

   if( isObject( %Obj.myPzone ) )
      %Obj.myPzone.delete();
}
```

C.16 Lesson #16 – MoveMap

In this short lesson, we will examine the ActionMap that comes with the prototype and discuss some small changes to it and other scripts that will ensure the behavior we are expecting from our game.

C.16.1. Required behavior.

In our game, we want the following key mappings:

W	Move forward.
A	Move left.
S	Move backward.
D	Move right.
SPACEBAR	Jump
Mouse Move	Camera yaw and pitch.
TAB	Switch 1st and 3rd POV.

One key mapping we would specifically like to disable (eventually) is:

ALT + C	Free camera mode.

We don't want people using free camera mode to cheat and find coins without risking their avatar's life.

C.16.2. MoveMap.

The Maze Runner prototype comes with an ActionMap already implementing the above mappings as well as many others. The name of this action map is MoveMap.

One of the things that new users find confusing is the fact that MoveMap is created in two places. It is created in the file

"\MazeRunner\prototype\client\scripts\default.bind.cs"

and in

"\MazeRunner\prototype\client\config.cs".

So, where do we go if we want to modify this ActionMap?

Well, if we look in the function initClient() in the file "\MazeRunner\prototype\client\init.cs", we will see this code:

```
   // Default player key bindings
   exec("./scripts/default.bind.cs");
   exec("./config.cs");
```

It seems pretty straightforward that we should edit "config.cs". But then, perhaps it isn't so straightforward.

The file "config.cs" is in fact saved every time we quit our game:

```
// In file: \MazeRunner\prototype\main.cs

   echo("Exporting client config");
   if (isObject(moveMap))
      moveMap.save("./client/config.cs", false);
```

This is why new folks get confused. With the MoveMap being created in two places and being automatically saved at the end of a game, one is left wondering. So, let me help clarify.

C.16.2.1. When to modify "config.cs".

If you are going to add a new key mapping to the MoveMap ActionMap, or if you are going to modify an existing mapping in the MoveMap ActionMap, you should be sure that you are not running the game and then make your changes in the "config.cs" file.

```
moveMap.bindCmd(keyboard, "y", "echo(\"Yo yo\");", "echo(\".. awesome yo.\");");
```

C.16.2.2. When to modify "default.bind.cs".

Never add new mappings or modify existing mappings (in MoveMap) in this file. The only things that should go into this file are

- alternate ActionMap definitions

```
GlobalActionMap.bind(keyboard, "F9", cycleDebugRenderMode);
```

- and functions that may be called from an action map.

```
function moveleft(%val)
{
   $mvLeftAction = %val * $movementSpeed;
}
```

C.16.3. Making our changes.

In our game, we are happy with current mappings, except that we wish to eventually disable free camera mode. So, when we want to do this, simply remove this line from "config.cs":

```
moveMap.bind(keyboard, "alt c", toggleCamera);
```

For now, I suggest leaving this in, but when we get ready to release our game to the public this line needs to be removed. Additionally, we might want to remove this code from "default.bind.cs":

```
function toggleCamera(%val)
{
   if (%val)
      commandToServer('ToggleCamera');
}
```

C.17 Lesson #17 – Level Loader

In this lesson, we will discuss the level loader code. We will not be listing all of the code here as it is rather lengthy. Instead, we will describe how it works and how the code is structured.

Please note, the level loader is responsible for loading and starting all elements of the level. This includes the fireball shooting block, which we have not completely covered yet. Specifically, we have not discussed the fireballs themselves, nor have we spoken, of the code that fires them. If you wish, you may skip ahead to lesson #20 in Chapter 11 to see how they work. Not doing so will not affect the current lesson, but until we complete that lesson, the level loader won't start the fireball blocks correctly.

C.17.1. Copy required files.

From the accompanying disk, please copy the file

"\MazeRunner\Lesson_012\teleporters.cs"

into

"\MazeRunner\prototype\server\scripts\MazeRunner".

Now, edit the function onServerCreated() in the file file: "\MazeRunner\prototype\server\game.cs" to look like this (BOLD lines are new or modified):

```
exec("./MazeRunner/teleporters.cs"); // MazeRunner
exec("./MazeRunner/levelloader.cs"); // MazeRunner
```

C.17.2. Levels versus layers.

In the following description, we will be using the words *level* and *layer*. A level comprises one or more layers of game elements. A level may have any number of layers.

C.17.3. Level files.

The premise of this level loader is quite simple. Our goal is to load a single mission and then, at any time we wish, load the components that make up a level. By using a level map and a level loader we may define as many levels as we want without needing to hand create an entire mission and then load the mission (which is generally slower than placing items by script for single-player games).

The first thing we need to do is define the parts of a level file.

C.17.3.1. Level file format.

We want to be able to make levels with multiple elements and multiple layers. To do this, the level file cannot be constrained to a fixed length. Instead, it must by dynamic.

After some thought I came up with the syntax for this file shown in Table C.17.1.

Table C.17.1. Line/Element Tokens.

Line/Element	Meaning
Line 0	This line is used to store the numeric ID of the level that follows this one.
LAYER_UP	This will increment the current elevation at which blocks and other elements are being placed by 4 meters.
LAYER_DOWN	This will decrement the current elevation at which blocks and other elements are being placed by 4 meters.
LAYER_DEFINE	This indicates to the level loader that the next line in the file will specify a layer type.
BLOCKS	This indicates to the level loader that the next 16 lines will contain a map that designates where blocks are placed.
OBSTACLES	This indicates to the level loader that the next 16 lines will contain a map that designates where obstacles (teleport stations and fireball blocks) are placed.
PICKUPS	This indicates to the level loader that the next 16 lines will contain a map that designates where pickups (coins) are placed.
PLAYERDROPS	This indicates to the level loader that the next 16 lines will contain a map that designates where the player will be dropped at the beginning of the mission.

That is it for the syntax. Now, let's designate what letters mean in the actual layer definitions (those 16 lines).

C.17.3.2. Tokens.

Each layer definition is composed of 16 lines of 16 characters, meaning that each layer definition may have up to 256 elements in it. Because we have a multitude of things to place (and because we want to leave room for expansion), we will be reusing letters (tokens) between layer types. Table C.17.2 lists what individual tokens mean in the various layer contexts.

Table C.17.2. Definitions of tokens.

Layer Type/Token	Meaning
BLOCKS	
A-J	These designate one of the level blocks we discussed in lesson #5.
0-9	These are the fade blocks. The number specifies the number of seconds until the block fades. We will discuss how this fading works in the next chapter.
OBSTACLES	
X , Y , Z	One of these will produce a teleport station.
0-9	These are the fireball blocks, where each number is a block firing in a specified direction. For example: 0 – North, 1 – NorthEast, ..., 7 – East , 8 – NorthEast, and 9 is random.
PICKUPS	
C	A coin.
PLAYERDROPS	
P	A player drop point. The player is dropped at the first point found.

C.17.4. Level loader mechanics.

The mechanics of the loader are pretty straightforward. It will consume whatever file it has been told to load until it has placed all of the contents or until it hits some kind of error in the level file.

C.17.5. Level loader definition.

So, we have some rules upon which to base our level building, and thus we have rules upon which to base the design of the loader. Furthermore, we know the loader must read the file until it is consumed, regardless of how many layers are defined in the file. Let's get started.

Go ahead and load up the "levelloader.cs" file in your favorite browser and then follow along as we discuss it here.

C.17.5.1. Elevations and level increments.

The first thing we do in our loader is define some global variables with which to track:

- **$BaseElevation** – Beginning elevation for every new level (not layer).
- **$LevelIncrement** – Level up/down step size.
- **$CurrentElevation** – Current elevation we are building at (current layer).

C.17.5.2. Classifying tokens.

We are dealing with a lot of different tokens. We will need to categorize these tokens into groups to minimize our code size. Because we don't want to waste time doing multiple comparisons to determine if any one token is a teleporter, a fireball block, etc., we need a way to reduce the effort required to categorize tokens. The trick is to create an array where the index of the array is the expected token and the value in the array gives us the information we need. For example:

```
$BLOCKCLASS[A] = NORMAL;
$BLOCKCLASS[B] = NORMAL;
$BLOCKCLASS[C] = NORMAL;
// ...

$BLOCKCLASS[0] = FADE;
$BLOCKCLASS[1] = FADE;
$BLOCKCLASS[2] = FADE;
// ...
```

In the above code, we're saying that any token "A..C" will correspond to a normal block while 0..2 will be a fade block.

So, we don't want to of write the code like this:

```
if( ( %token $= A ) || ( %token $= B ) || ( %token $= C ) )
{
   // Normal Block Code Here
}
else if if( ( %token $= 0 ) || ( %token $= 1 ) || ( %token $= 2 ) )
{
   // Fade Block Code Here
}
```

We can write it like this instead:

```
switch$( $BLOCKCLASS[%token] )
{
case NORMAL:
   // Normal Block Code Here

case FADE:
   // Fade Block Code Here
}
```

As you can see, this code is not only more elegant, but also significantly faster than the multiple comparison case before (and that was with only 6 of the 20 possible block cases shown).

C.17.5.3. BuildLevel().

We've prepared the globals and helper variables we'll need now let's write the loader function.

The buildLevel() function takes a single argument containing the numeric ID of the level to load. The function assumes that all level files are stored in the directory

"\MazeRunner\prototype\data\Missions\LevelMaps"

Given the number 0, the loader will attempt to open a file named

"\MazeRunner\prototype\data\Missions\LevelMaps\levelNum0.txt"

If the level loader successfully opens this file, the first thing it will do is read the first line, which contains the numeric ID of the level that follows this one. If no next level is defined, the loader fails out.

So far, nothing mysterious has been done, but the next bit of code may seem strange. For several lines, you will see bits of code like this:

```
if( isObject( gameLevelGroup )  )
   gameLevelGroup.delete();
MissionGroup.add( new SimGroup( gameLevelGroup ) );

gameLevelGroup.add( new SimGroup( mazeBlocksGroup ) );
gameLevelGroup.add( new SimGroup( fadeGroup ) );
gameLevelGroup.add( new SimGroup( FireBallMarkersGroup ) );
gameLevelGroup.add( new SimGroup( TeleportStationGroupX ) );
gameLevelGroup.add( new SimGroup( TeleportStationGroupY ) );
gameLevelGroup.add( new SimGroup( TeleportStationGroupZ ) );
gameLevelGroup.add( new SimGroup( TeleportStationEffectsGroup ) );
gameLevelGroup.add( new SimGroup( CoinsGroup ) );
```

Remember, we are building our levels dynamically. As part of this effort, we are destroying the prior level if it exists. Also, to make our lives simple, we will be tracking all of our objects in named SimGroups. This is ideal because much of the processing our game does is of an iterative nature, and it is easy to iterate over a SimGroup.

So, the above statement and the remainder like it in the function are merely removing the last level's SimGroups (if they exist) and then creating these named SimGroups:

- **gameLevelGroup** – This is the big daddy of all level SimGroups. It will contain all of our subsequent SimGroups for this level. Thus, deleting just this group kills all the children groups and their contents.
- **mazeBlocksGroup** – All normal blocks are stored in this group.
- **fadeGroup** – All fade-able blocks are stored here. Later we will iterate over this group to maintain the fadeblocks' behaviors.
- **fireBallMarkersGroup** – All fireBall blocks are stored here. Like the fadeGroup, we iterate over this group to keep the fireBall blocks firing.
- **TeleportStationX .. TeleportStationZ** – These three sets are used to store the three types of teleporter. Teleport stations stored in the same group will target each other.
- **TeleportStationEffectsGroup** – Although we have an onRemove() method that deletes the Pzone and Particle Emitter Node attached to a trigger when the trigger is deleted, I prefer to store the IDs of these effects in a SimGroup, too. That way, there is no question that they will get deleted on level load (or on mission exit).
- **CoinsGroup** – This last group stores all coins (that have not been picked up). We will use this later to determine when the level is complete and it is time to load the next one.

Next, we will see the beginning of the level loader's main processing loop:

```
   while(!%file.isEOF() )
   {
```

From this point on, the level loader will read in lines from the file until the file is empty or an error occurs.

Upon first entering this loop, the function reads a line and checks to see what task it represents, LAYER_UP, LAYER_DOWN, or LAYER_DEFINE. For a LAYER_UP or a LAYER_DOWN, we increment/decrement and then go back to the top of the loop (by using continue) to get the next task. If the task does not match any known task type, the function aborts.

Eventually, the task we get will be a LAYER_DEFINE. This tells the loader that the next line will be a LAYER_TYPE. So, the level loader reads the next line and decides which layer type it is, BLOCKS, OBSTACLES, PICKUPS, or PLAYERDROPS. If it is none of these, the function fails out.

Assuming the function read a valid layer type, it will use another of those system scripts supplied with the prototype and load the next 16 lines into an arrayObject (a scripted class I created so that we may create arrays that can be passed between functions and methods).

After reading in the next 16 lines (into our arrayObject), the level loader will then pass this array to a specialized function that does the layout for that level type:

- **layOutBlocks()** - This lays out normal blocks and fade blocks.
- **layOutObstacles()** - This lays out fireball blocks and teleport stations.
- **layOutPickups()** - This lays out coins.
- **playerDrop()** - This will drop the player into the level at a specified point.

After the current layout pass, the loader goes back to the top of the file-reading loop and continues until the end of file (or error).

Eventually, the file will be empty and we will drop out of the loop. At this point in the code, you will see a function (in two places) that may not yet be familiar to you:

```
if( fadeGroup.getCount() )
   fadeGroup.schedule( 5000 , fadePass );

if( FireBallMarkersGroup.getCount() )
   FireBallMarkersGroup.schedule( 1500 , firePass );
```

In both of the above statements, the script is telling the engine schedule an event to occur in, one in 5000 milliseconds and one in 1500 milliseconds.

The first event is the calling of the function fadePass(). It will be called like this:

```
   fadeGroup.fadePass();
```

The second event is the calling of the function firePass(). It will be called like this:

```
   FireBallMarkersGroup.firePass();
```

Each of these statements will cause a special function (not yet covered) to iterate over the specified SimGroup and to "do something" to each entry. We will cover this in the next chapter.

C.17.5.4. LayOutBlocks().

We will talk about the first of the four layout functions, and then I will leave you to examine the other three on your own. They are all designed quite similarly so there should be no mystery.

This function has the responsibility for creating content in the world based on the values in the arrayObject it has been passed.

To do its jobs, the function uses a nested loop and reads every token of every line and parses these tokens by category (using the trick we discussed at the beginning of this lesson) and then by specific type.

It is assumed that every token represents a world space of "4 4 4". Thus, the current position is incremented by "4 4 0" to keep us on the current layer.

When a token is found that matches a known category, an object in that category is created. Being smart, we named our files and datablocks in such a way that we can merely append the token to a generic version of the filename or datablock name when loading a file or building an object from a datablock.

Once this function has consumed all of the lines in the arrayObject, it deletes the object.

C.17.6. Temporary spawn point.

One side effect of destroying a level is that the player will fall into the lava because there is no place to stand. So, to solve this problem, while the level loads, we should create a place for the player to stand temporarily. This can be accomplished by editing the file

"\MazeRunner\prototype\data\Missions\mazerunner.mis"

and adding the following to the end of the mission file (BOLD lines are new):

```
   new TSStatic() {
      position = "0 0 295";
      rotation = "1 0 0 0";
      scale = "1 1 1";
      shapeName = "~/data/MazeRunner/Shapes/MazeBlock/blockA.dts";
   };
};
//--- OBJECT WRITE END ---
```

Additionally, to get the spawn to work properly, we need to move the spawn point. So, in the same file, locate our spawn point and change the position to this:

```
new SimGroup(PlayerDropPoints) {

   new SpawnSphere() {
      position = "0 0 300";
      rotation = "1 0 0 0";
      scale = "1 1 1";
      dataBlock = "SpawnSphereMarker";
      radius = "1";
      sphereWeight = "100";
      indoorWeight = "100";
      outdoorWeight = "100";
         locked = "False";
         lockCount = "0";
         homingCount = "0";
   };
};
```

Now, when we start the mission, the player will be high above the cauldron and on subsequent loads the avatar will be moved here temporarily.

C.17.7. Testing the level loader.

At this point, you should be able to test the level loader. Simply start the prototype, open the "Maze Runner" mission, open the console, and type "buildLevel(0);" Your player should be moved to the extra block we just inserted for a few seconds, and then it should drop onto a new level.

C.18 Lesson #18 – Game Events

In this lesson, we will examine the scripts used to fade blocks in and out, and we will examine the functions used to shoot fireballs on a regular basis. Now that we have covered the scheduling, string manipulation, and scripted math, we should be ready to examine how these gameplay scripts work.

Please note: This lesson depends on lesson #4.

C.18.1. Fade blocks.

There are three blocks of code we are interested in for the fade blocks. The first of these is in the file

"/MazeRunner/prototype/server/scripts/MazeRunner/levelloader.cs"

At the end of the function BuildLevel(), there is a little snippet of code that checks to see if there are any fade blocks in the fadeGroup SimGroup. If there are, the loader schedules a fadePass() in 5000 milliseconds:

```
   if( fadeGroup.getCount() )
      fadeGroup.schedule( 5000 , fadePass );
```

C.18.1.1. fadePass().

This function has the task of coming back every $stepTime (1000) milliseconds and updating all of the fade blocks. The motivation for updating all the blocks simultaneously is that it gives us greater control over the behavior of the blocks than if each block scheduled its own maintenance. Also, by maintaining a single entry and exit point, we only use one schedule, thus reducing overhead.

```
function SimSet::fadePass( %theSet )
{
   %theSet.forEach( fadeStep , true );

   %theSet.schedule( $stepTime  , fadePass );
}
```

As can be seen, this function merely iterates over the blocks in the set and runs fadeStep() on each of them.

C.18.1.2. FadeStep().

This function has the responsibility for advancing the fade status of an individual fade block by one time period. A fade block can be in one of three states:

1. waitToFadeOut – The block is waiting to begin a fade.
2. waitToFadeIn – The block is faded out and waiting to begin fading in.
3. wait – The block is in a dead-cycle waiting for all other blocks to complete the current fade cycle.

A fade cycle is always 10 seconds long (as implemented in "fadeblocks.cs"). During a single fade cycle, every single fade block will fade out, fade in, and wait for its peers to finish their fade cycle.

By using this method instead of allowing blocks to fade in, fade out, fade in, ad infinitum, without synchronizing, we avoid chaos. The game would be no fun if the blocks faded in and out chaotically. But, because we can rely on a cycle always taking 10 seconds and then repeating itself, the player can plan ahead after observing a cycle or two.

However, enough talking. Let's look at the code:

```
function StaticShape::fadeStep( %theBlock )
{
   %theBlock.timer = %theBlock.timer - $stepTime;

   // Check for flip-time
   if( %theBlock.timer <= 0 )
   {

      switch$(%theBlock.action)
      {
         case "waitToFadeOut":
            %theBlock.timer = $basePauseTime;
            %theBlock.startFade( $fadeTime , 0 , true );
            %theBlock.schedule( $fadeTime , setHidden , true );
            %theBlock.action  = "waitToFadeIn";

         case "waitToFadeIn":
            %theBlock.timer = $basePauseTime;
            %theBlock.setHidden( false );
            %theBlock.startFade( $fadeTime , 0 , false );
            %theBlock.action  = "wait";

         case "wait":
            %theBlock.timer = %Obj.maxTime;
            %theBlock.action  = "waitToFadeOut";
      }
   }
}
```

As we can see, individual blocks have an internal timer containing some predefined value. When that timer gets down to (or below) zero, it is time to change the block's state and to do some work.

Initially, all blocks will have these values:

- timer – This value will be between 1000 and 10,000 milliseconds.
- maxTime – This value will be the same as timer. The value in this field is never changed after the block is implemented.
- action – All blocks start out executing the action waitToFadeOut.

Now, if we restrict our discussion to just one block and assume that the block has a timer and maxTime of 1000 milliseconds, over time, we will see the behavior described in Table C.18.1.

Table C.18.1. Fade behavior of one block.

Time (ms)	Action(s)
0	- timer = timer – 1000 (0 <= 0 continue executing) **(block is visible)** - action == "waitToFadeOut" - Block starts to fade out. - Block schedules a hide. - action = "waitToFadeIn" - timer = 10000
1000	- timer = timer – 1000 (9000 > 0 skip) **(block is invisible)**
2000	- timer = timer – 1000 (8000 > 0 skip) **(block is invisible)**
3000	- timer = timer – 1000 (7000 > 0 skip) **(block is invisible)**
4000	- timer = timer – 1000 (6000 > 0 skip) **(block is invisible)**
5000	- timer = timer – 1000 (5000 > 0 skip) **(block is invisible)**
6000	- timer = timer – 1000 (4000 > 0 skip) **(block is invisible)**
7000	- timer = timer – 1000 (3000 > 0 skip) **(block is invisible)**
8000	- timer = timer – 1000 (2000 > 0 skip) **(block is invisible)**
9000	- timer = timer – 1000 (1000 > 0 skip) **(block is invisible)**
10000	- timer = timer – 1000 (0 <= 0 continue executing) **(block is invisible)** - action == "waitToFadeIn" - Block unhides. - Block starts to fade in. - action = "wait" - timer = 1000
11000	- timer = timer – 1000 (0 <= 0 continue executing) **(block is visible)** - action == "wait" - timer = 1000
...	Sequence repeats

The important thing to note about this behavior is that the fade blocks support up to ten blocks with incrementing (by 1000 milliseconds) fade times to be placed in order. Subsequently, these blocks will fade out in order. Then, one second after the last block fades out, the first block will start to fade back in. Thus, the fade in-and-out is deterministic and cyclic, allowing a player to observe a pattern and to memorize it.

C.18.2. Fire balls.

There are three blocks of code we are interested in for the fireball blocks. The first of these is in the file

"/MazeRunner/prototype/server/scripts/MazeRunner/levelloader.cs"

At the end of the function BuildLevel(), there is a little snippet of code that checks to see if there are any fireball blocks in the FireBallMarkersGroup SimGroup. If there are, the loader schedules a firePass() in 5000 milliseconds:

```
if( FireBallMarkersGroup.getCount() )
   FireBallMarkersGroup.schedule( 5000 , firePass );
```

C.18.2.1. firePass().

This function has the task of coming back every $stepTime (1000) milliseconds and checking each fireball block to see if that fireball block should fire a new fireball. Again, controlling fireballs this way (as with fade blocks) allows us to use a single schedule event to handle all of our fireball blocks. This is easy to understand, and efficient.

```
function SimSet::firePass( %theSet )
{
   %theSet.forEach( doFire , true );

   %theSet.schedule( $fireTime  , doFire );
}
```

C.18.2.2. doFire().

Again, we have created a function that will operate on individual blocks to enact each block's action if it is time to do so. Here is a summarized listing of the function:

```
function StaticShape::doFire( %marker ) {

   if( isObject( %marker.bullet ) ) return;

   // Handle random fire marker case
   %firePath = ( %marker.type == 9 ) ? getRandom( 0 , 9 ) :  %marker.type ;

    switch( %firePath )
    {
    //
    // NORTH
    //
    case 0:
        %marker.shootFireBall( FireBallProjectile , "0 1 0" , 20 );


    // ... similar code for case 1 .. 7


    //
    // DOWN
    //
    case 8:
        %marker.shootFireBall( FireBallProjectile , "0 0 -1" , 20 );

    }
}
```

We have not examined the shootFireBall() method, but when this method executes, it will create a projectile and store the ID of that projectile in the block's bullet field. When a projectile strikes an object, the projectile will explode and then self-delete.

So, our doFire() method first checks to see if this block has a bullet by seeing if the value in the bullet field is still an object. If it is, then we do not yet need to fire another bullet and the method exits.

If there is no current bullet, the method will next check to see if this is a random block. In the case that this block shoots in a random direction, it will get a random value between 0 and 8 and then continue.

Having selected a firing direction (or going with the fixed direction) we now enter a long case statement that shoots a new fireball by calling shootFireBall() and passing in the following information (in this order):

- **Projectile datablock** – This is the projectile to shoot.
- **Direction** – This is the direction to shoot in.
- **Velocity** – This is the velocity we want the fireball to move with.

Please note, we will examine the method shootFireball() in lesson #20.

C.19 Lesson #19 – FireBall Explosion

In this lesson, we will examine three datablocks that are supplied with the prototype. These datablocks are used to implement the explosion that occurs when a projectile (see lesson #20) explodes.

If you look in file "\MazeRunner\prototype\server\scripts\MazeRunner\FireBall.cs", you will find three datablocks:

- **FireBallExplosionParticle** – This datablock defines the particles that are used in the explosion.
- **FireBallExplosionEmitter** – This datablock defines the pattern for the explosion emission.
- **FireBallExplosion** – This datablock defines the way in which the emitter is played and the effects that the explosion has on the surroundings.

C.19.1. FireBallExplosionParticle.

Let's look at the code for this emitter:

```
datablock ParticleData(FireBallExplosionParticle : baseSmokePD0 )
{
   lifetimeMS           = 750;
   lifetimeVarianceMS   = 200;

   colors[0]      = "1 0.2 0.2 1.0";
   colors[1]      = "1.0 0.6 0.2 0.0";

   sizes[0]       = 1.5;
   sizes[1]       = 3.5;

};
```

We will first notice that it is inheriting from datablock baseSmokePD0. This is very important for the following reasons:

1. A large variety of effects can be created using a small set of particle textures.
2. The prototype included with this guide comes with a variety of predefined particle datablocks as well as emitters. You should use these as the base (through inheritance or good old cut-copy-paste) for your own particle effects and tweak just the parts that you need.

3. A large variety of effects can be created using a small set of particle textures. Yes, I just said this, but I want to drive the point home. You don't need to go crazy and create a ton of textures. Instead, tweak the datablock fields, and you will be surprised at the number of effects you can achieve.

In this case, we are inheriting a basic smoke particle and then adjusting the fields in Table C.19.1.

Table C.19.1. Fields being adjusted.

Fields	Purpose of Change
lifeTimeMS **lifeTimeVarianceMS**	The base particle has a rather long life, but we want our explosion particles to live for a shorter time.
colors[0] **colors[1]**	We're trying to get a reddish explosion that fades to a dark orange.
sizes[0] **sizes[1]**	The particle should start off fairly big and rapidly grow to a little more than double its original size.

C.19.2. FireBallExplosionEmitter.

Next, we must define an emitter. In this case, our emitter is new and does not inherit from a base emitter.

```
datablock ParticleEmitterData(FireBallExplosionEmitter)
{
   ejectionPeriodMS = 7;
   periodVarianceMS = 0;
   ejectionVelocity = 1;
   velocityVariance = 1;
   ejectionOffset   = 0;
   thetaMin         = 0;
   thetaMax         = 60;
   phiReferenceVel  = 0;
   phiVariance      = 360;
   overrideAdvances = false;
   particles = "FireBallExplosionParticle";
};
```

The above datablock will produce an emitter that will create a large number of particles in a short period. These particles will be ejected at between 1 and 2 meters per second with no offset. The direction of the emitter will vary from straight up to just above horizontal. Additionally, particles will be ejected in a complete circle about the up-vector at the point of explosion. Lastly, this emitter uses the particle we just defined.

C.19.3. FireBallExplosion.

This last datablock uses the prior two to define the actual explosion:

```
datablock ExplosionData(FireBallExplosion)
{
   lifeTimeMS = 2000;
   particleEmitter = FireBallExplosionEmitter;
   particleDensity = 50;
   particleRadius = 0.2;
   faceViewer     = true;

   // Dynamic light
   lightStartRadius = 0;
   lightEndRadius   = 6;
   lightStartColor  = "1 0.2 1";
   lightEndColor    = "1 0.6 0.2";
};
```

This explosion will live for 2 seconds, emitting particles the entire time. It uses the emitter we just defined and limits the number of simultaneous particles to just 50 at any one time. It varies the point of ejection randomly by up to 0.2 meter about the point of explosion. The particles are made to face the viewer at all times, thus making sure that the clouds of particles are always nice and uniform. The explosion will produce light in a 6 meter radius that starts off reddish and ends a dark orange. Please note, because the blocks are self-illuminating, this effect will not be very visible. You may wish to re-export the blocks without self-illumination enabled to see if the effect is more pleasing this way.

C.20 Lesson #20 – The FireBall

In this lesson, we will examine three of the six datablocks that are supplied with the prototype. These datablocks are used to implement the projectile representing the fireball.

If you look in the file: "\MazeRunner\prototype\server\scripts\MazeRunner\FireBall.cs", you will find three datablocks:

- **FireBallParticle** – This datablock defines the particles that are used for the projectile's trail.
- **FireBallEmitter** – This datablock defines the pattern for the trail.
- **FireBallProjectile** – This datablock defines the projectile itself and uses the above two datablocks as well as the three we discussed in lesson #19 (FireBallExplosionParticle, FireBallExplosionEmitter, and FireBallExplosion) which are used for the explosion.

C.20.1. FireBallParticle.

Again, we have chosen to implement our particle datablock by using inheritance, but this time many parameters have been modified:

```
datablock ParticleData(FireBallParticle : baseSmokePD0 )
{
   dragCoeffiecient     = 0.0;
   gravityCoefficient   = 0.0;
   inheritedVelFactor   = 0.0;

   lifetimeMS           = 350;
   lifetimeVarianceMS   = 50;

   spinRandomMin = -30.0;
   spinRandomMax = 30.0;

   colors[0]     = "1 0.7 0.7 1.0";
   colors[1]     = "1 0.7 0.7 1.0";
   colors[2]     = "1 0.7 0.7  0";
   sizes[0]      = 0.5;
   sizes[1]      = 0.7;
   sizes[2]      = 1.0;
   times[0]      = 0.0;
   times[1]      = 0.3;
   times[2]      = 1.0;
};
```

The particles this produces will not be affected by drag or by gravity, nor will they inherit any velocity from the emitter. This means, they will just hang in the air where they are produced.

They have a pretty long lifetime between 300 and 400 milliseconds.

As they hang in the air, they will spin back-and-forth between minus 30 and 30 degrees.

Lastly, the smoke will start as medium sized off-white poofs and end as large gauzy white poofs.

C.20.2. FireBallEmitter.

The emitter datablock is fairly short because it doesn't have a lot to do for smoke trails:

```
datablock ParticleEmitterData(FireBallEmitter)
{
   ejectionPeriodMS = 20;
   periodVarianceMS = 5;

   ejectionVelocity = 0.25;
   velocityVariance = 0.10;

   thetaMin         = 0.0;
   thetaMax         = 180.0;

   particles        = FireBallParticle;
};
```

This emitter will produce a new particle every 15 to 25 milliseconds meaning that the trail may be a little spotty (the projectile is moving at 20 meters per second if you will recall from lesson #18).

The particles themselves have very little velocity when ejected, and they are all ejected between straight up and straight down (we could make this range smaller to create a more narrow trail).

Lastly, the emitter uses the particle datablock we just discussed.

C.20.3. FireBallProjectile.

This datablock brings all of the work in the prior lesson and this one together to create the fireball:

```
datablock ProjectileData(FireBallProjectile)
{
   projectileShapeName =
      "~/data/MazeRunner/Shapes/Projectiles/projectile.dts";

   explosion          = FireBallExplosion;

   particleEmitter    = FireBallEmitter;

   armingDelay        = 0;

   lifetime           = 5000;
   fadeDelay          = 4800;

   isBallistic        = false;
};
```

This particle uses a mesh that is provided with the prototype. It is nothing more than a very small elongated pyramid with a simple texture applied (Figure C.20.1).

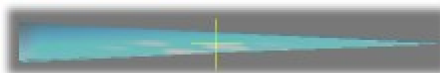


Figure C.20.1. Fireball projectile.

It uses the explosion datablock and the (smoke trail) emitter defined above.

There is no arming delay, so the projectile will explode as soon as it strikes an object.

The projectile will live for 5 seconds and begin to fade at 4.8 seconds. At the end of its lifetime, it will automatically be deleted if it has not already impacted upon something.

It is nonballisitic and will travel in a straight line along the path on which it is fired.

C.20.4. ShootFireBall().

We deferred our discussion of the fireball shooting method until this chapter so we would have the proper context. The main thing to understand is that when we create a projectile and put it into the world, it starts with an instantaneous velocity and direction (as specified at creation time):

```
function StaticShape::shootFireBall( %marker, %projectile , %pointingVector ,
%velocity)
{
    %bullet = new Projectile() {
        dataBlock        = %projectile;

        initialVelocity  =
           vectorScale( vectorNormalize(%pointingVector) ,
                        %velocity );

        initialPosition  = %marker.getWorldBoxCenter();

        sourceObject     = -1;
        sourceSlot       = -1;
        theMarker        = %marker;
    };

    %marker.bullet = %bullet;
    MissionCleanup.add(%bullet);
}
```

The most important things to see in the above code are:

1. The initial velocity is a combination of a direction and a magnitude.
2. The projectile can have any initialPosition, and we are choosing the centroid of the FireBall block. This is important, because it demonstrates that collision detection only occurs for penetrations of a collision mesh, not for objects or rays leaving the mesh, as is the case with this projectile.

C.21 Lesson #21 – Game Sounds

In this lesson, we will examine the different methods available to create audioDescription and audioProfile objects. This work will subsequently be used in Section 14.7 "Finishing the Prototype" to add sound to our game interfaces and game world.

For our game, we will need audioDescriptions and audioProfiles to play the following sounds:

- **Splash Screen Music** – We'd like to add some music to our splash screen when it is shown. This is a non-networked non-looping 2D sound.
- **Button-Over and Button Press Sounds For Main Menu** – We want our buttons to provide feedback when the mouse hovers over them and when we click on of them. These are both non-networked non-looping 2D sounds.
- **In-game Music** – We'd like some background music while playing our game, preferably a ambient loop of some sort. This is a non-networked looping 2D sound.
- **Fireball Firing and Explosion Sound** – It doesn't make much sense for our fireball blocks to shoot a fireball silently, and the explosion when the fireball collides with something should not be silent either. These are both networked non-looping 3D sounds.

C.21.1. The Audio Descriptions

In order to create audio profiles, we need to create audio descriptions first. Why? Because, the audioProfile object uses the audioDescription object.

In our list (above), we have three non-networked non-looping 2D sounds, one non-networked looping 2D sound, and a two networked non-looping 3D sounds. In total, this equates to a requirement for three different audio descriptions.

C.21.1.1. Non-Networked Non-Looping 2D Audio Description

```
new AudioDescription( MazeRunnerNonLooping2DADObj )
{
   volume            = 1.0;

   isLooping         = false;

   is3D              = false;

   type              = $GuiAudioType;
};
```

Using the *new* keyword, we have created an instance of AudioDescription descriptively named "MazeRunnerNonLooping2DADObj". Audio profiles using this description have the following attributes:

- Is non-networked. It is a normal object, not a datablock.
- Plays at full volume for the channel the sound is using.
- Is non-looping.
- Is non-3D.
- Is assigned to the $GUIAudioType channel and will thus be attenuated by changes to that channel.

C.21.1.2. Non-Networked Looping 2D Audio Description

```
new AudioDescription( MazeRunnerLooping2DADObj )
{
   volume            = 1.0;

   isLooping         = true;

   loopCount         = -1;

   is3D              = false;

   type              = $GuiAudioType;
};
```

Using the *new* keyword, we have created an instance of AudioDescription descriptively named "MazeRunnerLooping2DADObj". Audio profiles using this description have the following attributes:

- Is non-networked. It is a normal object, not a datablock.
- Plays at full volume for the channel the sound is using.
- Is looping.
- Loops infinitely. (We assigned -1 to loopCount, but we could have left it unspecified as well since the default value is -1.)
- Is non-3D.
- Is assigned to the $GUIAudioType channel and will thus be attenuated by changes to that channel.

C.21.1.3. Networked Non-Looping 3D Audio Description

```
datablock AudioDescription( MazeRunnerNonLooping3DADDB )
{
   volume            = 1.0;

   isLooping         = false;

   is3D              = true;

   ReferenceDistance = 2.0;

   MaxDistance       = 20.0;

   type              = $SimAudioType;
};
```

Using the *datablock* keyword, we have created an instance of AudioDescription descriptively named "MazeRunnerNonLooping3DADDB". Audio profiles using this description have the following attributes:

- Is networked. It is a datablock.
- Plays at full volume for the channel the sound is using.
- Is non-looping.
- Is 3D.
- Plays at max volume between 0 and 2 world units and attenuates to nearly zero at a distance of 20 world units, from the source position of the 3D sound.
- Is assigned to the $SimAudioType channel and will thus be attenuated by changes to that channel.

C.21.2. The Audio Profiles

Now that we have our three audio descriptions, we can create our audio profiles. In this case, we need one each for the sounds, but since several of these sounds, are similar except for the sound file played, we will only examine one from each category.

C.21.2.1. The Non-Looping GUI Sounds (Splash Screen and Buttons)

```
new AudioProfile(MazeRunnerGGSplashScreen)
{
   filename    = "~/data/GPGTBase/sound/gui/GGstartup.ogg";
   description = MazeRunnerNonLooping2DADObj;
};
```

Using the *new* keyword, we have created an instance of AudioProfile descriptively named "MazeRunnerGGSplashScreen". This audio profile will be used when the GarageGames Splash Screen is shown and has the following attributes:

- It plays the GarageGames startup sound from the demo kit. (This sound file was renamed to GGStartup.ogg from startup.ogg and included with GPGT base data for your use).
- It uses our Non-Looping 2D Audio Description Object: "MazeRunnerNonLooping2DADObj".

C.21.2.2. The Looping GUI Sound (In-game Music)

```
new AudioProfile(MazeRunnerLevelLoop)
{
   filename    = "~/data/GPGTBase/sound/gui/levelLoop.ogg";
   description = MazeRunnerLooping2DADObj;
};
```

Using the *new* keyword, we have created an instance of AudioProfile descriptively named "MazeRunnerLevelLoop". This audio profile will be used for in-game music and has the following attributes:

- It plays a short ambient loop provided on the accompanying disk.
- It uses our Looping 2D Audio Description Object: "MazeRunnerLooping2DADObj".

C.21.2.3. The Networked Sounds (Fireball Firing and Explosion)

```
datablock AudioProfile(MazeRunnerFireballExplosionSound)
{
   filename    = "~/data/GPGTBase/sound/GenericExplosionSound.ogg";
   description = MazeRunnerNonLooping3DADDB;
};
```

Using the *datablock* keyword, we have created an instance of AudioProfile descriptively named "MazeRunnerFireballExplosionSound". This audio profile will be used for the sound effect attached to a fireball explosion and has the following attributes:

- It plays a generic explosion sound that is included on the accompanying disk for your use. This sound is derived from the Crossbow_explosion.ogg found in the TGE Demo.
- It uses our Non-Looping 3D Audio Description Datablock: "MazeRunnerNonLooping3DADDB".

C.21.3. Using The Audio Profiles

All of the above audio descriptions and audio profiles are provided on the accompanying disk. We will be using them later when we follow the instructions in Section 14.7 "Finishing the Prototype". However, the question of use should at least be addressed. How does one use these new sounds?

The sounds we created are used in three ways:

1. Attached to a GUI Control – The button over and press sounds above will be used by a GUI button control. As you will discover while reading Chapter 12, this attachment is achieved using GUI profiles.

2. Attached to a special effect – Our explosion sound is used by the explosion object. As we saw in Section 11.3 "Explosions", we can assign an AudioProfile datablock to the ExplosionData *soundProfile* field. When an explosion is created with this datablock it will automatically play the sound specified by our AudioProfile datablock.

```
datablock ExplosionData( FireballExplosion )
{
   // ...

   soundProfile = MazeRunnerFireballExplosionSound;

   // ...
};
```

3. Played Manually – Lastly, we can play sounds manually. We simple call alxPlay() and pass it the name or ID of a non-networked 2D sound AudioProfile.

```
// Play the GG Splash Screen Sound
alxPlay( MazeRunnerGGSplashScreen );
```

C.22 Finishing the Prototype

Thus far, you have probably been working your way through the guide, learning about various features of the Torque Game Engine. Along the way, we have stopped to do little lessons that created one or more game elements to be used in the game.

At this point we don't really have a playable game. We have just a short distance to go before our game reaches the playable prototype stage. To get our game ready for play testing we must do the following two things:

1. **Finish gameplay code.** At this point, we can start the Maze Runner mission and then manually load a level, but our player doesn't get moved to the right spot on the level and there is pretty much no interaction. We need to change this. Specifically, we need to make the levels load automatically, have the player die when struck by a fire ball or after falling into the lava, load the next level when all the coins are collected, and aware our player with a new life on a successful level completion.
2. **Improve feedback.** With the final mechanics in place, we need to provide just a little bit more feedback to the player. Specifically, we need to update the playGUI to show how many lives we have, how many coins we've collected (score), and how many coins are left for a level. Also, while we are about this, we will add sounds for the fire ball firing and explosions, and then add some GUI sounds and music to make if feel like a whole package.

C.23 Finish Gameplay Code

By this point you should be feeling pretty comfortable with TorqueScript and with navigating the prototype directory structure. So, the kid gloves are coming off. In the next few pages, we will run through some terse discussions. We will examine newly added scripts and modifications to scripts we discussed in prior lessons.

C.23.1. Copy Required Files

Before we continue, please:

1. Copy "\MazeRunner\MazeRunner_Post_Finishing_the_Prototype\prototype2" into "\MazeRunner\".
2. Copy "\MazeRunner\MazeRunner_Post_Finishing_the_Prototype\main.cs" into "\MazeRunner\".

The new "main.cs" file points to the newly added "prototype2" mod director. "prototype2" contains all of the change we are about to discuss and is ready to play if would like to try it before continuing.

C.23.2. Breaking the Law

The first thing we will do is break the law. OK, we're not breaking the law, but we are doing something that I warned you *NOT* to do earlier. Namely, we are going to make a global variable for tracking the ID of the player. Then, we are going to use it to implement gameplay scripts, and later to keep our interfaces up to date.

We are, in effect, ignoring the client-server divide. This is both good and bad. It is good because it makes writing the scripts for our single-player game simple. It is bad because it ties us to a singleplayer game *ONLY*. If later we decide to make this game support multiple players we will experience at least some pain while we modify our scripts to handle this new mode.

So, why are we doing this? Well, first I know that in this book we will only ever play this game in singleplayer mode. Second, the game is simple enough that later, if you do convert this to multiplayer, the pain won't be too bad and it will serve as an excellent object lesson in making good decisions.

Excuses and reason aside, we must implement this change. To do so, I have modified the method GameConnection::createPlayer() in “game.cs” to look like this (BOLD lines are newcode):

```
function GameConnection::createPlayer(%this, %spawnPoint)
{
   // Create the player object
   %player = new Player()
   {
      dataBlock = MazeRunner; // Change this line
      client = %this;
   };
   MissionCleanup.add(%player);

   $Game::Player = %player; //MazeRunner
```

Now, whenever we want the player's ID, we can just reference the '$Game::Player' global.

C.23.3. Automatic Startup

To this point, we have been manually loading missions by typing "buildLevel(0);". That is just fine for testing purposes, but we really need the game to load when the mission is loading.

C.23.3.1. Experiments in loading.

If we examine the "game.cs" file closely, we will see that it has a variety of functions and methods. Among these are some promising-sounding places to put a script for automatically loading our first level:

- **onMissionLoaded()** - Hmmm... this sounds good. The mission is loaded, so we should be good to go.
- **startGame()** - This sounds good, too. I mean we do want to start the game, right?
- **GameConnection::createPlayer()** - OK, maybe you wouldn't think of this one. This is a hint, actually.

Great, we have some possible places to do the level loading, but what are the steps we need to follow in order to load our level?

Can we simply put a buildLevel() call in one of these? Why don't we try it?

Add this code to the end of onMissionLoaded() (BOLD lines are new code):

```
      startGame();

      buildLevel(0);
   }
```

After restarting the game and reloading the mission, this may work, or it may work partially, or the game may hang. It depends.

At this point in the game startup process, there is some ambiguity in timing due to latencies that can vary from run to run. This means that any of the following actions can occur:

1. The game starts correctly and the player is on the correct spawn point. This is what we want. Unfortunately, this doesn't always happen.
2. The level loads and the player gets dropped on the safe spawn point, end of story. Now we're stuck.
3. If timing conspires against you, all the resources that need to have been loaded won't be ready and the loading code will just hang. This is the worst possibility.

So, what is happening here? Well, the mission was loaded, but the player had not been created yet, so our scripts for moving the player can't work. They have no object to move. (If you're curious, you can see the player moving script by looking at the playerDrop() function in "levelloader.cs".)

Since putting buildLevel() after startGame() didn't work, that pretty much rules out our placing the function call in startGame() too. What about GameConnection::createPlayer() then?

Let's try that next:

```
   %this.player = %player;
   %this.setControlObject(%player);

   BuildLevel(0); //MazeRunner
}
```

Perfect! This is guaranteed to work properly every time. The mission is always loaded prior to the player being created, so the scripts have valid object IDs to work with.

C.23.4. Dying

Another problem with our prior revision of this game was that we didn't get killed by the lava or fireballs. Let's remedy that now.

C.23.4.1. KillZone.

To be killed by the lava, we need some way to know we're in it. Now, we could make our water block into a lava block by changing the water type. However, as part of our game design, we chose to make the player invincible, so this won't really help. I mean, we could in theory make our player have a very low damage level, make it damageable, and then maybe, just maybe falling in the lava would kill him.

The thing is we don't really want the player to die. We just want to lose a life and move to the spawn point. When a player dies (getState() returns "dead"), the player will no longer move or take move commands until it is replaced with a new instance. This is by design and is not what we want in this instance.

So, long story short, we get creative. Let's create a really big trigger (named KillZone) and place it in the lava. Then, we can just write an onEnterTrigger() callback that will take away a life and move us to the spawn point. Perfect!

```
datablock TriggerData(KillZoneTrigger)
{
   tickPeriodMS = 100;
};

function KillZoneTrigger::onEnterTrigger(%DB , %Trigger , %Obj)
{
   %Obj.loseALife();
}
```

The above code defines the datablock for this trigger, and the callback calls the method loseALife() (described below) on the object entering the trigger, but what about placement? This code will do the placement:

```
function buildKillZone()
{
   new Trigger(KillZone) {
      position   = "-256 256 40";
      rotation   = "1 0 0 0";
      scale      = "512 512 25";
      dataBlock  = "KillZoneTrigger";
      polyhedron = "0.0000000 0.0000000 0.0000000 1.0000000 0.0000000 0.0000000
0.0000000 -1.0000000 0.0000000 0.0000000 0.0000000 1.0000000";
   };

   MissionGroup.add( KillZone );
}
```

Then we can add a call to this code in onMissionLoaded() to do the creation (BOLD lines are new code):

```
function onMissionLoaded()
{
   buildKillZone(); // MazeRunner

   startGame();
}
```

So, what about that loseALife() thing?

C.23.4.2. Player::loseALife().

The easiest way to handle removing lives is to make a method scoped to the Player class (so it can be called on the Player object) that handles all of the bookkeeping. This simplifies things greatly. Yes, right now only two things can kill the player, but later you might add more, and having killing code all over the place would be very bad.

Here is the code (located in “mazerunnerplayer.cs"):

```
function Player::loseALife( %player )
{
   // 1
   %player.lives--;

   // 2
   if( %player.lives <= 0 )
   {
      schedule( 0 , 0 , endGame );
      return;
   }

   // 3
   %player.setVelocity("0 0 0");
   %player.setTransform(%player.spawnPointTransform);
}
```

This code does the following:

1. It decrements the player's life counter. (Yes, we haven't talked about this yet. It's coming up soon.)
2. It checks to see if all of our lives are gone and then schedules a call to endGame() (in “game.cs”) to unload the mission, destroy the player, disconnect the client from the server, and get us back into the main menu. You may wonder why we don't call endGame() directly.
3. If the game is not over, the player is moved back to its last spawn point. This information is stored in the player by playerDrop() in “levelloader.cs":

```
$Game::Player.spawnPointTransform = (%actX SPC %actY SPC $CurrentElevation);
```

C.23.4.3. Initial lives.

In order to take away lives, we must have lives to take. The best place to add initial lives to the player is either in its onAdd() method, or at the location where we create it. I chose the onAdd() method (in "mazerunnerplayer.cs"; BOLD lines are new code):

```
function MazeRunner::onAdd( %DB , %Obj )
{
   Parent::onAdd( %DB , %Obj );
   %Obj.lives = 3;
}
```

C.23.4.4. Fireballs.

OK, we got a little off topic there, but we're back now. The next question is; how do fireballs kill?

The projectile object has an onCollision() callback that is called for collisions with any world object. So, if we write a version of this callback in the namespace of our projectile, we can have that callback check to see if the player was hit and call loseALife():

```
function FireBallProjectile::onCollision( %projectileDB ,
                                          %projectileObj ,
                                          %collidedObj ,
                                          %fade , %vec , %speed )
{
   if (%collidedObj.getClassName() $= "Player")
   {
      %collidedObj.loseALife();
   }
}
```

In the above callback (located in "fireballs.cs"), the engine is asked to get the class name for the collided-with object. It then compares this to "Player". If the comparison comes back true, loseALife() is called on the collided-with object.

Alternate solution #1.

There is an alternate way to write this code that would actually work in more cases (i.e., for Player and aiPlayer):

```
// Alternate implementation

function FireBallProjectile::onCollision( %projectileDB ,
                                          %projectileObj ,
                                          %collidedObj ,
                                          %fade , %vec , %speed )
{
   if (%collidedObj.getType() $= $TypeMasks::PlayerObjectType )
   {
      %collidedObj.loseALife();
   }
}
```

This alternate implementation uses the getType() method to get a bitmask for the collided-with object. The bitmask contains bit settings for all classes from which the object is derived as well as for the class itself. So, as I alluded to, if the collision occurred against an aiPlayer (which is derived from Player), this comparison would still work, whereas the prior code would not. In this game, we don't have that worry, so let's leave it as is.

Originally, as I wrote this code for the book, I was using a bleeding edge version of the engine (1.4 before release) and I ran into a bug (that has since been fixed) where %collidedObj was always getting "1". For a moment, I thought I was stuck. Then, it occurred to me that there are other ways to solve the identification problem, and I wrote this code:

```
%Offset = vectorSub( %vec , $Game::Player.getWorldBoxCenter() );
%Len = vectorLen( %offset );

if( %len < 1.7 )
{
    $Game::Player.loseALife();
}
```

This code used the position of the projectile's collision and then compared it to the position of the player's centroid. If the distance between them was small (1.7 meters or less), in all likelihood the object that had been hit was the player and I called loseALife(). This solved my temporary problem, and on the occasional time when the player wasn't hit but was just close to the collision point, the difference was not noticeable.

The lesson here is that TGE is very flexible and you can often solve the same problem in many ways. So, don't let one dead end stop you.

C.23.4.5. Out of lives.

At some time, after all this losing of lives, the player will be out of lives. According to our initial rules list, this means the game is up, time to go home. As we have already seen (above) the loseALife() method handles this case and ends the game for us.

C.23.5. Moving On

The last thing we need to fix with regards to gameplay is moving on to the next level and getting our extra life.

C.23.5.1. Last coin.

The rules stated that when the last coin is picked up, the current level should be unloaded and the next level should be loaded. So, how do we do this?

If you recall, the inventory system has a callback called onPickup(). When we discussed this callback, I said that you might want to override it to implement special behaviors. This is one of those times.

If you will look in "coins.cs", you will find this implementation of onPickup():

```
function Coin::onPickup( %pickupDB , %pickupObj , %ownerObj )
{
   // 1
   %status = Parent::onPickup( %pickupDB , %pickupObj , %ownerObj );

   // 2
   if (CoinsGroup.getCount() == 0 )
   {
      buildLevel($Game::NextLevelMap);

      $Game::Player.lives++;
   }

   // 3
   return %status;
}
```

This callback does the following:

1. It takes advantage of the previously written pickup code by calling the Parent:: version.
2. It then checks to see if the SimGroup CoinsGroup is empty. In the case that it is empty, buildLevel() is called with the stored numeric ID of the next level, and a new life is added to our player.
3. Last, but not least, it returns the return status from the Parent call. This is important because the method/callback that called onPickup() in the first place might care if the pickup was successful or not.

C.23.6. Gameplay Scripting Completed

We are officially done with the gameplay scripting now. The game is now in a playable state, and we could define some levels and ship it off to our testers at this point. If this were a business venture, that would be the plan, but since we're learning about Torque and not running a gaming business, let's continue.

C.24 Improve Feedback

To make the game easier to play, we should provide some information to the player so s/he knows how many lives are remaining, what the score is, and how many coins are left on a level. Also, adding sounds to the fireballs will make them a little easier to detect. Lastly, if we add some sounds and music we will have a nicely rounded prototype.

C.24.1. Copy Required Files

Before we continue, please:

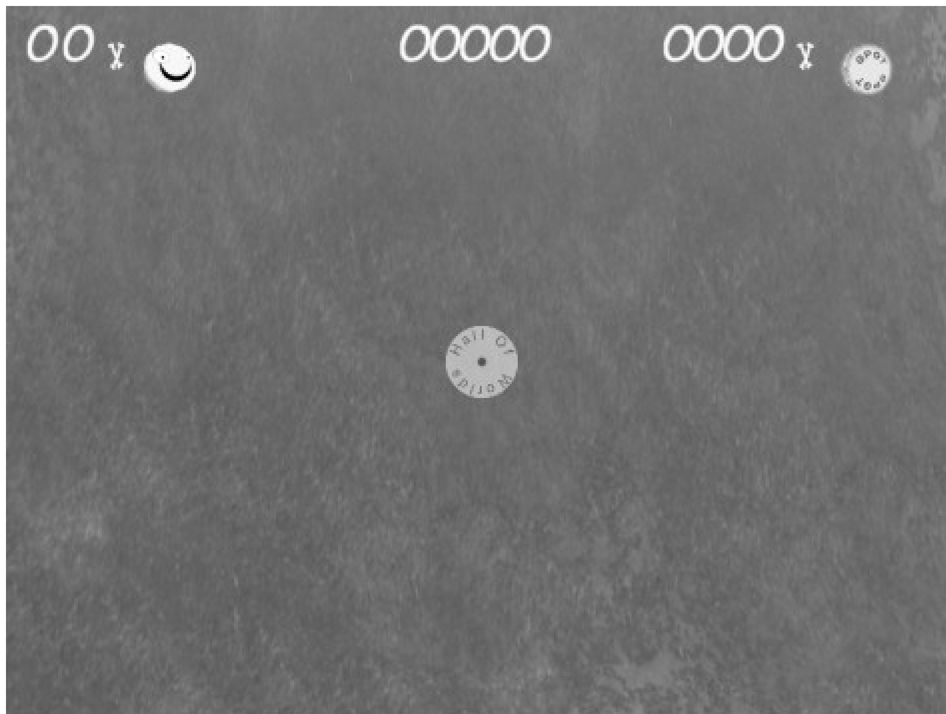
1. Copy "\MazeRunner\MazeRunner_Post_Improve_Feedback\prototype3" into "\MazeRunner\".
2. Copy "\MazeRunner\MazeRunner_Post_Improve_Feedback\main.cs" into "\MazeRunner\".

The new "main.cs" file points to the newly added "prototype3" mod director. "prototype3" contains all of the change we are about to discuss and is ready to play if would like to try it before continuing.

C.24.2. New playGUI HUDs

If you start the game and run the "Maze Runner" mission you will see that the new and improved playGUI has the three HUDS at the top of the screen (Figure C.24.1.)

Figure C.24.1 New HUDs.



The three HUDs are:

- **Lives Counter (Upper-Left)** – Shows number of lives the player has left.
- **Score (Upper-Middle)** – Shows number of coins thus far recovered.
- **Remaining coins for level (Upper-Right)** – Shows coins left till end of level.

These HUDS should look quite familiar. They are the same counters we discussed in Chapter 13, being put to good use in our prototype game.

To make your life easier I have created a completely new playGUI containing these HUDS and placed it and all the scripts and content associated with it in "~/client/ui/PlayGUIs/". To get this new playGUI interface loaded instead of the old one, I changed the initClient() function in "~/client/init.cs" as follows:

```
function initClient()
{
   // ...

   //exec("./ui/PlayGui.gui");       // Prior to Maze Runner
   exec("./ui/PlayGUIs/PlayGui.cs"); // MazeRunner (Load My GUI)

   // ...

   //exec("./scripts/playGui.cs"); // Prior to Maze Runner

   // ...
}
```

This change simply tells the function NOT to load the old "PlayGUI.gui" and "PlayGUI.cs" and to load my "PlayGUIs/PlayGui.cs" instead. This new script will automatically load the remainder of the scripts required to build the new playGUI.

Now, let's talk about how these HUDs are hooked up.

C.24.2.1. Hooking up the lives HUD.

The lives counter is initialized in MazeRunner::onAdd(), from "mazerunnerplayer.cs" (BOLD lines are new code):

```
function MazeRunner::onAdd( %DB , %Obj )
{
   Parent::onAdd( %DB , %Obj );
   %Obj.lives = 3;
   livescounter.setCounterValue(%Obj.lives);
}
```

It is decremented in Player::loseALife() , from "mazerunnerplayer.cs" (BOLD lines are new code):

```
function Player::loseALife( %player )
{
   // 1
   %player.lives--;
   livescounter.setCounterValue(%player.lives);

   // ...
}
```

It is incremented in Coin::onPickup(), from “coins.cs” (BOLD lines are new code):

```
function Coin::onPickup( %pickupDB , %pickupObj , %ownerObj )
{
   // ...

   if (CoinsGroup.getCount() == 0 )
   {
       // ...

      livescounter.setCounterValue($Game::Player.lives);
   }

   // ...
}
```

C.24.2.2. Hooking up the score HUD.

The score counter is initialized in GameConnection::createPlayer(), from “~\server\scripts\game.cs” (BOLD lines are new code):

```
function GameConnection::createPlayer(%this, %spawnPoint)
{
   // ...

   BuildLevel(0);
   scorecounter.setCounterValue(0);
}
```

It is incremented in Coin::onPickup(), from “coins.cs” (BOLD lines are newcode):

```
function Coin::onPickup( %pickupDB , %pickupObj , %ownerObj )
{
   // ...

   scorecounter.setCounterValue( scorecounter.getCounterValue() + 1 );

   // ...
}
```

C.24.2.3. Hooking up the remaining coins HUD.

The coins counter is initialized at the very end of buildLevel(), from “levelloader.cs” (BOLD lines are new code):

```
function BuildLevel( %levelNum )
{

   // ...

   coincounter.setCounterValue( CoinsGroup.getCount() );
}
```

It is decrement in Coin::onPickup(), from "coins.cs" (BOLD lines are new code):

```
function Coin::onPickup( %pickupDB , %pickupObj , %ownerObj )
{
   // ...

   coincounter.setCounterValue( CoinsGroup.getCount() );

   // ...
}
```

C.24.3. Adding Sounds

To give the game a little more pizazz and to make it feel more finished, we need to add a few sounds. As you will recall, in Chapter 11, we made several audio descriptions and audio profiles. I have included all of these and a few others in to separate places.

The 2D sound descriptions and profiles have been added to a new file named: "~\client\scripts\MazeRunnerGUISounds.cs". This includes:

- **MazeRunnerNonLooping2DADObj** – A non-looping 2D audioDescription object for use with audioProfile objects.
- **MazeRunnerLooping2DADObj** – A looping 2D audioDescription object for use with audioProfile objects.
- **MazeRunnerGGSplashScreen** – An audioProfile object to play music when the Garage Games splash screen is displayed.
- **MazeRunnerButtonOver and MazeRunnerButtonPress** – Two audioProfile objects used to play button over and press sounds.
- **MazeRunnerLevelLoop** – An audioProfile object used to play an ambient loop during game play.

This file is loaded by "~/client/init.cs" using this code:

```
   /// Load client-side Audio Profiles/Descriptions
   exec("./scripts/audioProfiles.cs");
   exec("./scripts/MazeRunnerGUISounds.cs"); // Maze Runner
```

The 3D sound descriptions and profiles have been added to the existing "fireballs.cs" file at the top and include:

- **MazeRunnerNonLooping3DADDB** – A non-looping 3D audioProfile datablock for use with audioProfile datablocks.
- **MazeRunnerFireballExplosionSound** – An audioProfile datablock that is played for each fireball when it is shot.
- **MazeRunnerFireballExplosionSound** – An audioProfile datablock that is used by the FireBallExplosion datablock to play and explosion sound.

These sounds will now be loaded when "fireballs.cs" is executed.

Now, let's briefly discuss how each of our new sounds is used.

C.24.3.1. Adding Sound To Splash Screen

The simplest way to add a sound to the Garage Games splash screen is to play the sound when the splash screen is displayed. If we look in the file "~/client/ui/StartupGui.gui", we will find a method named loadStartup(). This method is used to display the splash screen. To have the game play a sound when the splash screen is displayed, I made these changes:

```
function loadStartup()
{
   // ...

   //alxPlay(AudioStartup); //Before Maze Runner

   alxPlay(MazeRunnerGGSplashScreen); //Maze Runner
}
```

C.24.3.2. Adding Sound To Buttons

To have the menu buttons play a sound when the mouse passes over a button and when a button is clicked, I needed to define a new GuiControlProfile object and fill in the proper fields:

```
if(!isObject(MainMenuButtonProfile))
   new GuiControlProfile (MainMenuButtonProfile)
   {
       // ...

       soundButtonOver = "MazeRunnerButtonOver";

       soundButtonDown = "MazeRunnerButtonPress";
   };
```

I then made sure that each button in the main menu ("~/client/ui/mainMenuGui.gui") used this new profile.

```
  // ...
   new GuiButtonCtrl() {
      profile = "MainMenuButtonProfile";
   // ...
```

C.24.3.3. Adding Ambient Loop To Game

To add the ambient loop to our game I simply added an alxPlay() statement to the onWake() callback and a reciprocal alxStop() statement to the onSleep() statement for the new playGUI. Both of these callbacks are located in "~/client/ui/playGUIs/playGUI.cs" and now look like this:

```
function PlayGui::onWake( %this ) {
    $enableDirectInput = "1";

    activateDirectInput();

    // Activate the game's action map
    moveMap.push();

    %this.levelLoop = alxPlay(MazeRunnerLevelLoop); // Maze Runner
}
```

and this:

```
function PlayGui::onSleep( %this ) {
    // Pop the keymap
    moveMap.pop();

    if(isObject ( %this.levelLoop ) )
       alxStop(%this.levelLoop); // Maze Runner
}
```

Notice, that I simply stored the handle returned from alxPlay into a aptly named dynamic field "levelLoop" created on the fly in the playGUI control object. Later I checked to see if the handle represented a valid handle and stopped playing the sound associated with it using alxStop().

C.24.3.4. Playing Sounds When Fireballs Are Fired

To play the firing sound, we will again use the playAudio() ShapeBase method. Although we don't care in this singleplay game, by doing this, we insure that every client will hear the sound with no extra effort on our part. To do this, I modified the "StaticShape::shootFireBall()" console method to include this code:

```
function StaticShape::shootFireBall( %marker,
                                     %projectile ,
                                     %pointingVector ,
                                     %velocity)
{
   // ...
   %marker.playAudio( 0 , MazeRunnerFireballFiringSound );
}
```

If you will recall, all fireballs are fired from the center position of a fireball block's world box. Thus, we can approximate the correct location for the firing sound by simply playing the firing sound using the block that marks the origin of the shot itself. In this case I merely called playAudio() and played the "MazeRunnerFireballFiringSound" audioProfile datablock in sound slot 0.

C.24.3.5. Adding Explosion Sounds To An Explosion Datablock

The last sound that was added is the explosion sound. This was accomplished by assigning the new "MazeRunnerFireballExplosionSound" audioProfile datablock do the existing "FireBallExplosion" datablocks *soundProfile* field.

```
datablock ExplosionData(FireBallExplosion)
{
   // ...
   soundProfile = "MazeRunnerFireballExplosionSound";
   // ...
};
```

That's it, we now have a working prototype that we can distribute for testing. what's next?