



A Tutorial by Jamie Douglas

Contents

Folder Contents	Page 1
Introduction	Page 1
Part 1 – Add the Puzzles	Page 3
Part 2 – Add Baddies & Coins	Page 45
Part 3 – Coding the Ending	Page 50
Part 4 – Additional Bits	Page 54
Part 5 (optional) – Change the HUD	Page 55
Part 6 (optional) – Animated Intro screen	Page 57
Part 7 (optional) – Aqualung & Underwater	Page 59
Conclusion	Page 63

Folder Contents

Howto.pdf	–	This tutorial document, in Adobe PDF format.
Dizzy.exe	–	The game program
Setup.exe	–	The setup program
Dizzy.ini	–	A file which stores settings
Dizzy.txt	–	A game log file (created when the game is first run)
Readme.txt	–	The game 'readme' file
NewHUD.tga	–	A replacement .tga file (used in Part 5)
'data' folder	–	Containing all the files needed by the game.

Introduction

This tutorial game has been written to show you how to make a 'Dizzy' game using DizzyAGE by Alexandru Simion. The files in this .zip file are the basic game files included with DizzyAGE, with one exception: The 'map' file (found in **data -> map -> dizzy.map**) has been changed, and is a variant on the small 'Dizzy 3.5' game map.

Throughout this tutorial, I will assume that you are familiar with the Map Editor. If not, have a play around with it first, or watch my YouTube video called '[DizzyAGE Tutorial Part 1 – Map Editor Basics](#)'. It will also help if you watch the videos entitled '[DizzyAGE Tutorial Part 2 – More Editor Functions](#)' and '[DizzyAGE Tutorial Part 3 – Script Basics](#)'.

Additionally, everything you place in the map should have the SHADER property set to OPAQUE unless otherwise specified.

Once you are familiar with the editor, open up the [dizzy.map](#) file in the map editor, and you'll see the Dizzy 3.5 map displayed (Fig.1).

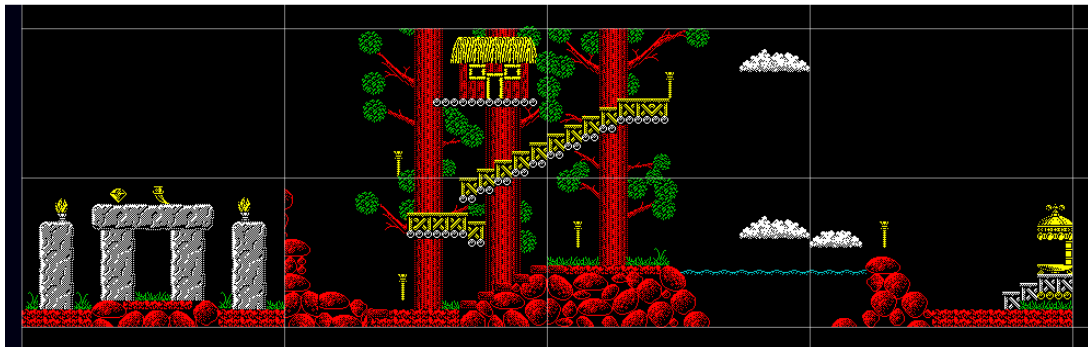


Fig. 1: Map Editor screen, with Dizzy 3.5 map loaded.

Click on the 'eye' icon (second from left) in the editor, and select 'material view'. The game screen will change to show which kinds of material are in the map (Fig.2)

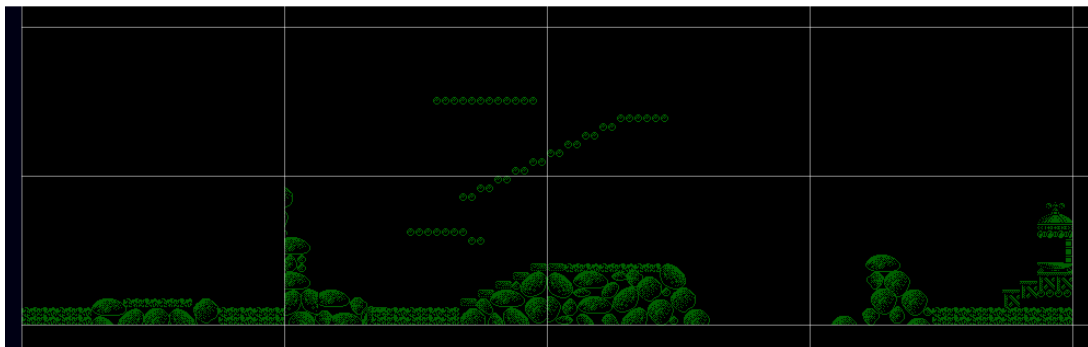


Fig. 2: Map Editor screen, in 'material' view

In this view, brushes with material property set as 'Block' (which 'blocks' you, meaning you can walk on it) are shown in green. Water is shown in Blue, and 'Cloud' materials are shown in Grey. Brushes with material property set as 'Air' are invisible in this view. Comparing Fig. 1 and 2, it is clear that the water brush and cloud brushes are currently set as 'Air' material.

Change the view back to 'default', and open up the 'scripts' folder ([data -> scripts](#)). Here you will see the scripts that contain all the code used by the game. Parts 1 – 4 of this tutorial will only use two script files: [game.gs](#) (where most of the coding will go), and [gamedef.gs](#) (where we will define some variables).

Part 1 – Coding the Puzzles

For this tutorial, we will be adding 10 puzzles. These are as follows:

- 1 Use Pickaxe on the Large Rock - Get past the Large Rock
- 2 Use the Oil on the Switch – Make the Lift move
- 3 Give the Ear Trumpet to Grand Dizzy - Get a Lump of Driftwood
- 4 Use the Lump of Driftwood on the Water
- 5 Use a Stick on the Driftwood - Make the Driftwood move
- 6 Give the Garden Trowel to Dylan - Get a Cupcake
- 7 Give the Cupcake to Dora - Get some Medicine
- 8 Give the Medicine to Grand Dizzy - Get some Cheese
- 9 Give the Cheese to the Rat - Get past the Rat
- 10 Use the Small Rock on the Button - Start the Transporter

We won't be coding the puzzles in the order shown above, as some puzzles will have very similar code.

A. Use the Pickaxe on the Large Rock

First of all, make sure you have the dizzy.map file open in the map editor.

Select tile 101 (boulders1), and select the top left boulder (Fig. 3). Use the 'S' key to toggle 'Snap to Grid' off and on if you need to.

Ensure the following properties are set:

TYPE: STATIC
MATERIAL: BLOCK
DRAW: IMG+MAT

Then change the colour to red, change the ID property to **2000**, and place it in the map as shown in Fig 4. This Brush (*brushes with property TYPE set as STATIC are known as 'Brushes' in the code, with DYNAMIC known as 'Objects' in the code*) will be the Large Rock in puzzle A, and will block the way.

Next, go back to tile 101 (Fig. 3), but this time select the small boulder at the bottom right corner. Change the colour to Red, and set the following properties:

TYPE: DYNAMIC
MATERIAL: AIR
DRAW: IMG+MAT
DISABLE: YES
CLASS: ITEM



Fig. 3: Selecting the 'Boulder' tile

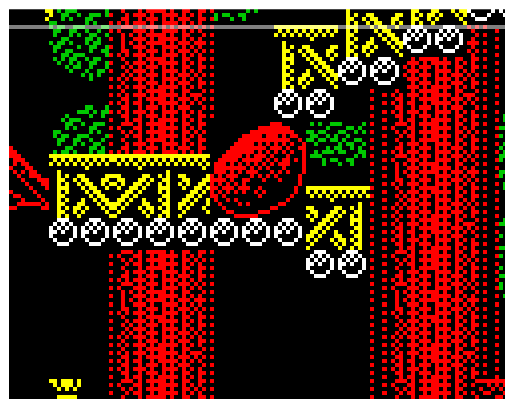


Fig. 4: Placing the Large Rock brush

Set the ID to **1000**, and place it in front of the larger boulder you've just placed down. (Fig. 5) I always try to put all items on layer **1** (to ensure they don't get accidentally hidden behind scenery, which is usually set as layer 0), so as this object has the CLASS property set as ITEM, change the LAYER to '**1**'. This will be the Small Rock item that will show once you smash the larger rock.

Next, open up tile 0 (default), which is a plain block of colour. Select a square of size 16x16, and place it on the map next to the large boulder (Fig. 5). Set the following properties:

TYPE: DYNAMIC
ID: 2001
MATERIAL: AIR
DRAW: MAT
DISABLE: NO
CLASS: ACTION

This will be the actionable 'Trigger' which calls the code for the Large Rock. Objects need to have TYPE property set to DYNAMIC to be actionable, and as the TYPE property for the rock is STATIC, it can't be actioned, so needs a 'Trigger' for Dizzy to action instead. I usually change the colour of 'Triggers' to Purple, which makes them stand out in the map. As the DRAW property has been set to MAT only, the image of it won't be shown in the game.

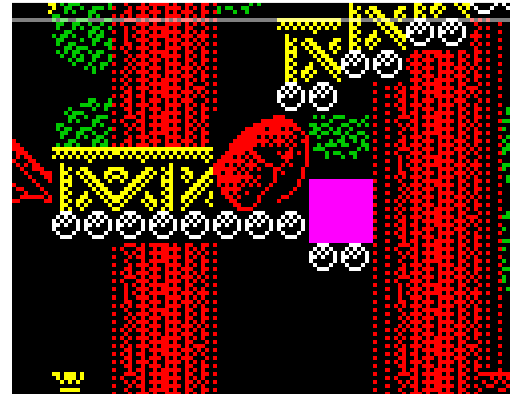


Fig. 5: Placing the small Rock item and the actionable 'Trigger'

Right-click on the Small Rock item and click 'pick brush'. This will create a duplicate of the Small Rock item for you to place down somewhere. Before you do however, open up the brush menu, open up tile 214 (items4), and select the Pickaxe in the top right corner. Leave the colour as Red, but change the ID to **1001**, and the DISABLE property to NO. Now place it down in the bottom left corner of the same room, on top of the rocks (Fig. 6). Don't forget to change the LAYER property to '**1**'.



Fig. 6: Placing the Pickaxe item

Now we have the following in the map, waiting to be coded (Don't forget to save the map!):

1000 Small Rock 'Item' Object
1001 Pickaxe 'Item' Object
2000 Large Rock 'Blocker' Brush
2001 Trigger 'Actionable' Object

Now that we have placed everything that is needed in the map for the first puzzle, we can start to code it!

Open up the script file '**game.gs**', and scroll to the bottom, just below the lines saying:

```

////////////////////////////////////
// Interactions
////////////////////////////////////

```

This is the best place to put the main coding for the game.

First, we need to write a new 'action object' function for the purple 'Trigger'. This is the code that the game will run when you press 'enter' while standing in front of it. We'll start with the basics:

```

////////////////////////////////////
//Large Rock
////////////////////////////////////
func ActionObject_2001()
{

}

```

The above code creates a function called **ActionObject_2001()**. Each function requires **{** and **}** to specify the start and end of the code for that function. Note that the actionable 'Trigger' for the rock is ID **2001**. Of course, we need more code than this, so in between the **{** and **}**, put the following:

```

Message0(4,4,"THERE IS A LARGE\nBOULDER IN THE WAY");
MessagePop();

```

Line 1 displays a message (Message0 uses a red border) at co-ordinates 4,4 and with text of **THERE IS A LARGE BOULDER IN THE WAY**. The **\n** in the middle is used to tell the game that the following text will start on a new line. This is vital for ensuring that the message box won't be too wide for the screen.

Line 2 pops all message boxes when the player presses 'enter'.

Obviously, after we have read this message, we don't need to see it again. The way to do this is to change the STATUS property of the 'Trigger' object once the message has been read, and to check the STATUS properties each time the 'trigger' object is actioned. To do this, we need to insert the following code (highlighted in Red):

```

func ActionObject_2001()
{
    idx = ObjFind(2001);
    if(ObjGet(idx,O_STATUS)==0)
    {
        Message0(4,4,"THERE IS A LARGE\nBOULDER IN THE WAY");
        MessagePop();
        ObjSet(idx,O_STATUS,5);
    }
}

```

The Red code is the code needed to check the STATUS property of object **2001**, display the message, and then change the STATUS property. The STATUS property of each brush and object in the map is always **0** by default, unless changed by the user.

The first line of the red code defines variable **idx** first of all as **ObjFind(2001)**. This code gets the game to 'find' Object **2001** (the trigger) in the map, and store it in variable **idx** for use later on in the code. Note that if we later re-define variable **idx**, the previous value will be overwritten.

The second line of red code is an **if** statement, which gets the STATUS property of object **idx** (the variable we earlier defined as object **2001**), and checks if it is equal to **0**. If it is, the game runs the code contained in the following **{** and **}**. If not, that code is ignored, and the game jumps to the code immediately afterwards, of which there isn't any in this case, as it is the end of the function.

In this case, the STATUS property will be equal to **0**, so the message code (in black) will be run. Once the message code is run, the next red line of code will be run, which sets the STATUS property of object **idx** (the variable we earlier defined as object **2001**) to **5**. This ensures that the messages won't be shown the next time, and we can now add the following code, which determines what is done when the STATUS property is **5**:

```
func ActionObject_2001()
{
    idx = ObjFind(2001);
    if (ObjGet (idx,O_STATUS)==5)
    {
        idx = OpenDialogInventory();
        if (idx!=-1) UseObject (idx);
        return;
    }
    if (ObjGet (idx,O_STATUS)==0)
    {
        Message0 (4,4,"THERE IS A LARGE\nBOULDER IN THE WAY");
        MessagePop ();
        ObjSet (idx,O_STATUS,5);
    }
}
```

The first Red line of the code above is exactly the same as the **if** statement mentioned earlier, which gets the STATUS property of object **idx** (the variable we earlier defined as object **2001**). Except this time it checks if it is equal to **5**. If it is, the game then runs 3 lines of code. The first line of these opens the inventory dialogue window, and **re-defines variable idx** as the number of whichever item the player chooses from the inventory. If the player doesn't select an item, **idx** is re-defined as **-1**. The second line is a compressed **if** statement, telling the game that if variable **idx** is **anything other than -1** (in other words, if the player selected an item), then run function **UseObject(idx)**, with variable **idx** being carried over to that function. The third line is the

return; command, which stops the game from going any further into that function, stopping it from clashing with the following **if** statement.

Now that the code for the **ActionObject_2001()** function has been written, we can move onto the **UseObject_2001(idx)** function, which is called by the game when you want to use an item on an actionable object.

So underneath the **ActionObject_2001()** function, start a new function, as follows:

```
func UseObject_2001( idx )
{
    if (ObjGet (idx,O_ID)==1001)
    {
    }
    else
    {
        DropObject (idx);
    }
}
```

First things first, the **idx** variable specified in brackets in the function name **UseObject_2001(idx)** is the same variable as we carried over earlier using the code **UseObject(idx)**. This ensures that we have a variable in this function which has the item ID stored within it.

The second line of code is another **if** statement, which is checking if the ID property for variable **idx** (which is the item selected from inventory) is equal to **1001** (the ID of the **Pickaxe**). If it is, the game will run the code contained in the following **{** and **}**. If not (the **else** line of code), it will run the code in the **{** and **}** following the **else** statement. That line of code simply tells the game to drop object **idx**.

We now of course need to tell the game what to do if the object selected from the inventory is the **pickaxe** (ID **1001**). We do that by adding code, considering the following points: we need to display a message; remove the **pickaxe** from the inventory; disable the trigger (to stop the code being called again); disable the boulder, and enable the smaller **rock** object

First we'll display a message. We do this using the same code as before:

```
Message0 (6,5,"YOU USE THE PICKAXE\nTO SMASH THE BOULDER");
MessagePop();
```

Next, we should remove the **pickaxe** item from the inventory:

```
InventorySub (idx);
```

This removes the item stored as variable **idx** (currently **1001** – the **pickaxe**) from the inventory.

Next, we'll disable the actionable trigger object (ID **2001**), as we will no longer need it, and disabling it will stop the code being called later on:

```
idx = ObjFind(2001);  
ObjSet(idx,O_DISABLE,1);
```

The above code **re-defines** variable **idx** as object **2001** (the trigger) on the first line, then sets the DISABLE property of variable **idx** as **1** (YES).

Now we do the same for the **Small Rock** item (ID **1000**), except this time we're enabling it, so we set the DISABLE property of it to **0** (NO):

```
idx = ObjFind(1000);  
ObjSet(idx,O_DISABLE,0);
```

Finally, we'll disable the **Large Rock** (ID **2000**). However the Large Rock is a **Brush**, not an **Object**, so we have to do it a different way, as the DISABLE property won't override the MATERIAL property, which is set to BLOCK. We do it by using the DRAW property instead, which is currently set to **3** (IMG + MAT), which draws both the image of the brush, and the material (which is set to BLOCK). If we set the DRAW property to **0** (NONE), it will not be drawn either as an image or as a material, effectively 'disabling' it.

Because the Large Rock is a Brush, we also have to use different functions, so we use **BrushFind** instead of **ObjFind**, and **BrushSet** instead of **ObjSet**:

```
idx = BrushFind(2000);  
BrushSet(idx,B_DRAW,0);  
GameCommand(CMD_REFRESH);
```

So the code above **re-defines** variable **idx** as Brush **2000** (the large rock) on the first line, then sets the DRAW property of variable **idx** as **0** (NONE). Note also that because we're changing a **brush**, not an **object**, we need to use **B_** rather than **O_** when we set it.

The third line is needed to refresh the brushes in the game after we've changed one or more of them (which we have just done). If this line isn't included, the changes in the second line of code won't show up in the game. If changing two or more brushes at once, you only need to use this line once, as it will refresh all brushes at once.

So by the end of all that, you'll have a function which looks something like this:

```

func UseObject_2001( idx )
{
    if(ObjGet(idx,O_ID)==1001)
    {
        Message0(6,5,"YOU USE THE PICKAXE\nTO SMASH THE BOULDER");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(2001);
        ObjSet(idx,O_DISABLE,1);

        idx = ObjFind(1000);
        ObjSet(idx,O_DISABLE,0);

        idx = BrushFind(2000);
        BrushSet(idx,B_DRAW,0);
        GameCommand(CMD_REFRESH);
    }
    else
    {
        DropObject(idx);
    }
}

```

Don't forget to save the file! If you run the game to test it however, you'll find that Dizzy falls down through the water, and into the screen below! To fix this, we need to change his start position when the game first loads.

Open up file **gamedef.gs** and change the following lines:

```

#def PLAYER_BEGINX          640          // set the starting position x
of the player in map (when a new game begins)
#def PLAYER_BEGINY          246          // set the starting position y
of the player in map (when a new game begins)

```

To this (changes highlighted in Red):

```

#def PLAYER_BEGINX          332          // set the starting position x
of the player in map (when a new game begins)
#def PLAYER_BEGINY          382          // set the starting position y
of the player in map (when a new game begins)

```

This will change his position to land, and enable you to test the first puzzle. But you'll soon discover another problem. There is no name for either of the items! To set the names, you need to find the following code in **game.gs**:

```

func ObjectsSetNames()
{
    // ...
}

```

This is where the names are set, so remove the **// ...** from the function, and put the following two lines there instead:

```

ObjSetName(ObjFind(1000),"A HEAVY ROCK");
ObjSetName(ObjFind(1001),"A STURDY PICKAXE");

```

These lines find objects **1000** (the **Small Rock**) and **1001** (the **Pickaxe**), and set their names.

Save the file, start the game again, and you should now be able to get past the Large Rock.

B. Give the Cheese to the Rat

This isn't the next puzzle the player will have to get past, however it is very similar to the Rock puzzle we've just done, so it makes sense to explain how the code for the Rock puzzle can be copied and changed for the Rat puzzle.

For this puzzle, we will need 3 things: A Rat; a static 'blocker'; and the Cheese item. The Rat character will serve as the actionable 'trigger' object, and does not therefore need to be invisible. The 'blocker' will be a Static Brush, however unlike the Rock earlier, it will be invisible. Finally the Cheese item will be invisible, as the player will get it from Grand Dizzy when playing the game.

Go into the map, and hover over the purple 'trigger' object for the rock, that you placed earlier. Right-click, and a menu will open. Select 'pick brush', and you will find that you're now holding a duplicate of the purple 'Trigger' object. This will be the basis of the 'Rat' object, as most of the properties that the Rat object will need are already set. However there are a few that need to be changed.

The first is the tile it uses, so open up the list of tiles, and browse through them until you find the 'Rat' tile (190 – 'mouse'). Select this and your tile will change to that tile. You will notice that the Rat is purple, with a red border! This is nothing to worry about. The purple is because we haven't yet changed the colour, and the red border is because we currently have the 'SHADER' property set as OPAQUE instead of BLEND. Change the colour first, by going into the colour menu and selecting one. For this tutorial, I will change it to Grey.

Next, change the following properties of the Rat:

FLIP: FLIP X
SHADER: BLEND
DRAW: IMG+MAT

Setting the FLIP property to FLIP X will change the direction the Rat faces (on the X axis), and changing SHADER to BLEND will remove the red border, and 'blend' the Rat with the scenery behind. (Because parts of the Rat are coloured black, the Rat won't show as transparent when set as BLEND). Changing the DRAW property to IMG+MAT will draw it in the game.

Also change the ID property to **2002**, and then place the Rat in the map in the position shown in Fig 7.

Now select the Large Rock Brush in the same way you selected the purple 'Trigger' object (Right-click over it, then click 'pick brush'). This will form the basis of the 'blocker' Brush. Obviously this will need changing, so select the plain tile from the list of tiles (tile 0 - default), and select an 8x8 block. Change the following property of it:

DRAW: MAT

This sets the Brush to only display in the Material map. This means that although you won't be able to see it as an image (DRAW isn't set to IMG), the game will still take account of which Material it is.

Next, change the colour to Dark Blue (I always set invisible blockers to this colour, to help distinguish them), and change the ID property to **2003**. Place it in the map as shown in Fig 7, extending it down by dragging it as you place it.

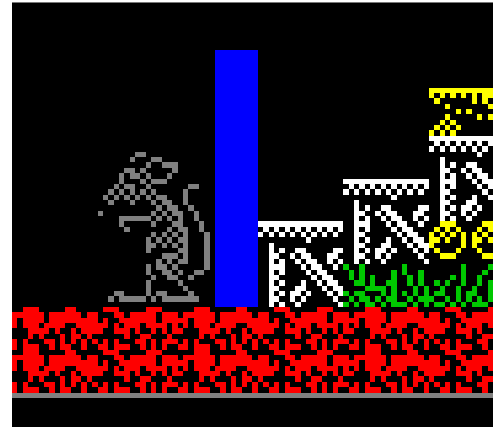


Fig. 7: Placing the actionable Rat object and the invisible 'blocker' brush.

Always ensure any 'blockers' you put in game maps are higher than 4 'blocks', as Dizzy can jump on top of anything that is 4 blocks high. Also take account of anything nearby that the player may be able to climb up and use to jump over your blocker.

Finally, we need to place the Cheese item into the map. This will be disabled to start with, so right-click and 'pick brush' on the Small Rock item (which is also disabled to start with, meaning less properties to change). Open up the tile menu, open up tile 218 (items8), and select the Cheese item in the middle. Change the colour to Yellow, and change the ID to **1009**. Place it down in the map next to the hut (Fig. 8), as this is where Grand Dizzy will be placed later (the player will get the Cheese item from Grand Dizzy). Don't forget to change the LAYER property of the Cheese to **'1'**.



Fig. 8: Placing the Cheese item next to the hut.

Now we can start coding the Rat. The code we use will be almost identical to the code we wrote for the Large Rock, as we are doing almost exactly the same things: Actioning a 'Trigger' item (in this case the Rat), which will display a message (in this case a conversation). Then when re-actioning it, opening the inventory, and when the correct item is selected, displaying another message (in this case a conversation), and removing the 'blocker'.

So because of this, we don't need to write all the code out again. We can simply copy and paste the code that we wrote for the Large Rock, then change parts of it. So first of all, copy the following function:

```

////////////////////////////////////
//Large Rock
////////////////////////////////////
func ActionObject_2001()
{
    idx = ObjFind(2001);
    if(ObjGet(idx,O_STATUS)==5)
    {
        idx = OpenDialogInventory();
        if(idx!=-1) UseObject(idx);
        return;
    }
    if(ObjGet(idx,O_STATUS)==0)
    {
        Message0(4,4,"THERE IS A LARGE\nBOULDER IN THE WAY");
        MessagePop();
        ObjSet(idx,O_STATUS,5);
    }
}

```

Paste it into the code below the Large Rock function, and change the following parts highlighted in Red:

```

////////////////////////////////////
//Rat
////////////////////////////////////
func ActionObject_2002()
{
    idx = ObjFind(2002);
    if(ObjGet(idx,O_STATUS)==5)
    {
        idx = OpenDialogInventory();
        if(idx!=-1) UseObject(idx);
        return;
    }
    if(ObjGet(idx,O_STATUS)==0)
    {
        Message1(4,4,"\"HELLO RAT, CAN\nYOU LET ME PAST?\"");
        Message2(6,8,"\"NO, I'M HUNGRY.\");
        MessagePop();
        ObjSet(idx,O_STATUS,5);
    }
}

```

Comparing the above, you can see just how little needs to be changed in the code. Just the ID number and the messages need to be changed to make the Large Rock code suitable for the Rat. In the message code, **Message1** is where Dizzy is talking; while **Message2** is where the other character is talking (**Message0** is narration). They all do the same thing, but with different border and text colours. In the messages, you can see that the two characters **"** are used either side of the speech. This is in order to show quotation marks in the message, which distinguishes it as speech.

Now we'll do a similar thing for the other part of the Large Rock code. So copy the second part of the Large Rock code, as follows:

```

func UseObject_2001( idx )
{
    if (ObjGet (idx,O_ID)==1001)
    {
        Message0 (6,5,"YOU USE THE PICKAXE\nTO SMASH THE BOULDER");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(2001);
        ObjSet (idx,O_DISABLE,1);

        idx = ObjFind(1000);
        ObjSet (idx,O_DISABLE,0);

        idx = BrushFind(2000);
        BrushSet (idx,B_DRAW,0);
        GameCommand (CMD_REFRESH);
    }
    else
    {
        DropObject (idx);
    }
}

```

Paste it below the other part of the Rat code, and change the following sections highlighted in Red:

```

func UseObject_2002( idx )
{
    if (ObjGet (idx,O_ID)==1009)
    {
        Message0 (6,5,"YOU GIVE THE\nCHEESE TO THE RAT");
        Message2 (6,8,"\"MMM THANK YOU!\nYOU MAY PASS.\");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(2002);
        ObjSet (idx,O_CLASS,0);

        idx = BrushFind(2003);
        BrushSet (idx,B_DRAW,0);
        GameCommand (CMD_REFRESH);
    }
    else
    {
        DropObject (idx);
    }
}

```

The **ID** and **Message** changes are the same as that explained above. Lower down, the **CLASS** property of Object **2002** (the Rat) is changed as we don't want to disable the Rat, and changing the **CLASS** to **0** (NONE) will stop the player actioning it again. We have removed the two lines of code that relate to **Object 1000**, because in the case of the Rat puzzle, there is no other item to enable.

Finally, don't forget to set the name of the Cheese item in function **ObjectsSetNames()**:

```
ObjSetName(ObjFind(1009),"SOME SMELLY CHEESE");
```

If you wish to test this puzzle now, you can change the DISABLE property of the Cheese item to NO, and test it (You'll have to use DEV mode to get past the water though). Don't forget to change the property back to YES though!

C. Use the Lump of Driftwood on the Water

This puzzle will be the first of two puzzles enabling the player to get over the water. This part will put the Driftwood into the water. The second part (Puzzle D, below) will deal with getting the Driftwood to move back and forth.

As with the two previous puzzles, this one starts out fairly similar, in that we will use an item on an actionable 'Trigger' object, which will then enable another Object (rather than disabling a 'blocker' Brush).

First of all then, right-click and 'pick brush' on the purple 'Trigger' object for the Large Rock. This has the exact same properties that we need for the Water 'Trigger' Object, and all we need to do is change the ID property to **2004**. Place this in the map next to the water, as shown in Fig 9.



Fig. 9: Placing the actionable 'Trigger' object next to the water

Now we need to place the Driftwood item in the map. This will be disabled to start with, so right-click and 'pick brush' on the Cheese item (which is also disabled to start with). Open up the brush menu, open up tile 132 (decorations3), and select the 16x16 square shown in Fig 10. Change the colour to a darker shade of Red, and change the ID to **1004**. Also change the FLIP property of it to FLIP XY, so that it is upside down. Place it down in the map on the other side of the hut to the Cheese (Fig. 11), as the

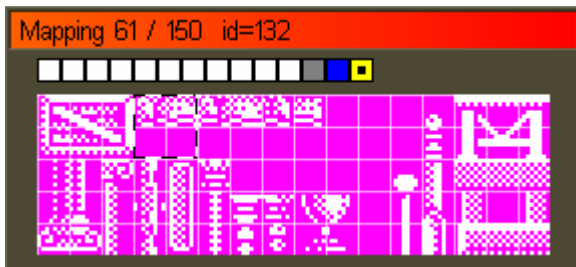


Fig. 10: Selecting the 'Driftwood' item

hut is where Grand Dizzy will be placed later (the player will get the Driftwood item from Grand Dizzy as well as the Cheese). Don't forget to change the LAYER property of the Driftwood to **'1'**.

Before we start coding this puzzle however, we need one more thing that neither of the two previous puzzles had. That is another Object (not an item) which will be enabled after using the Driftwood item on the 'Trigger' object. This Object will be the Driftwood that shows up in the water, showing that the player has dropped the Driftwood into the water.

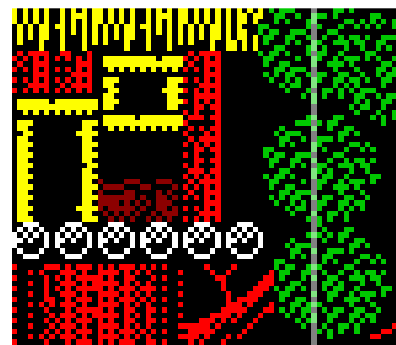


Fig. 11: Placing the Driftwood item in the map.

So to place this, right-click and 'pick brush' on the purple 'Trigger' object, open up the brush menu, open up tile 125 (woods), and select the smaller piece of driftwood which is under the very long piece at the top. Again, change the colour to the same Dark Red that you used for the Driftwood item, set the ID property of it to **2005**, and change the following properties:

DRAW: IMG+MAT
DISABLE: YES
CLASS: NONE
COLLIDER: HARD COLLISION

The COLLIDER property HARD COLLISION ensures that the object can be stood on top of, and the DRAW property is changed to show both Image and Material in the game. The CLASS property is changed to NONE because it is simply an object that the player will ride on.

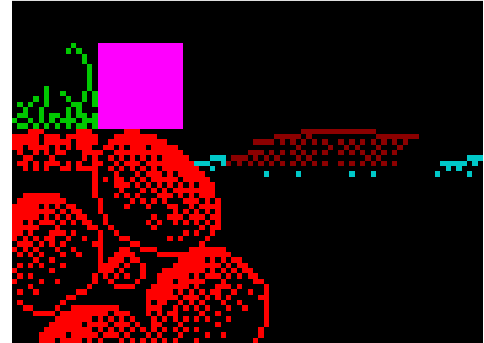


Fig. 12: Placing the Driftwood object in the map.

Now place it down in the map in the position shown in Fig 12, and set the LAYER property to '**1**' so that it is always in front of the Water. Now we can write the code for it.

As with the Rat code, this code will turn out to be very similar, and will only need a few things altering. So again, copy the first part of the Large Rock code, and paste it below the Rat code. Now change the following sections of it, as highlighted in Red:

```
////////////////////////////////////  
//Driftwood  
////////////////////////////////////  
func ActionObject_2004()  
{  
    idx = ObjFind(2004);  
    if(ObjGet(idx,O_STATUS)==5)  
    {  
        idx = OpenDialogInventory();  
        if(idx!=-1) UseObject(idx);  
        return;  
    }  
    if(ObjGet(idx,O_STATUS)==0)  
    {  
        Message1(4,4,"\"I NEED TO GET\nACROSS THE WATER\"");  
        MessagePop();  
        ObjSet(idx,O_STATUS,5);  
    }  
}
```

As you can see, it again only needs minor modifications to the ID and the Message to be suitable for this puzzle.

The same will be true of the second part of the code. So again, copy the second part of the Large Rock code, and paste it below the first part of the Driftwood code.

Now change the following sections of it, as highlighted in Red:

```
func UseObject_2004( idx )
{
    if(ObjGet(idx,O_ID)==1004)
    {
        Message0(6,5,"YOU DROP THE DRIFTWOOD\nINTO THE WATER");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(2005);
        ObjSet(idx,O_DISABLE,0);
    }
    else
    {
        DropObject(idx);
    }
}
```

Note that we've removed the code for disabling the 'Trigger' object, and also removed the code for removing a 'blocker' Brush. Removing the 'blocker' Brush code is obviously because there isn't a blocker in this puzzle, and removing the code for disabling the 'Trigger' object is because we will use the 'Trigger' again for another puzzle, so still need it enabled. We've also enabled the 'Driftwood' Object (ID 2005) instead of an item.

Don't forget to add the item name to function **ObjectsSetNames()**:

```
ObjSetName(ObjFind(1004),"A LUMP OF DRIFTWOOD");
```

If you want to test this puzzle, you'll again have to set the `DISABLE` property of the Driftwood item to `NO` so that you can test it. If you do test it, you'll notice that the Driftwood stays still in the water, rather than bobbing up and down. So we'll add a bit of simple code to enable it to move up and down.

For this, we are going to write our own function. Don't panic, it's nothing to worry about! We'll start from the beginning, and everything will be explained. If however you don't want to do this just yet, you can skip to Puzzle D below.

To make the bobbing look realistic, we want more than a simple 'up, up, down down' movement. So we'll write it so that from the bottom, it moves up 2, then up 1, then down 1, then down 2, then repeats. This will create the impression that it is 'bobbing', as it will move less at the top of the 'bob'.

So to start with, we need to create that function. Start by putting the following code into the bottom of file **game.js**:

```
// Bob an Object up and down
```

```
func Move_Inwater(obj)
{
}
```

Simple eh? The variable **obj** that is specified in brackets will contain the ID number of the object we want to move. By carrying the ID number over in this way, we can tell the function which Object to move.

Ok, now we'll add a bit more to the function. First things first, we need to tell the game to find in the map the ID number that we stored as variable **obj**, then store that as another variable, which we'll call **idx**:

```
idx = ObjFind(obj);
```

Now, we don't want the game to run through the whole function if the object is disabled – there's no point – so we'll add a bit of code in between the **{** and **}**:

```
if(ObjGet(idx,O_DISABLE)) return;
```

This checks if Object **idx** has the DISABLE property as YES (**1**), and stops the function if it does (by using **return;**). You'll notice that there are no **{** and **}** brackets in this **if** statement. This is because if there is only one piece of code after an **if** statement, it doesn't need the **{** and **}** brackets. It is only if there is more than one piece of code after an **if** statement that it needs them.

Also notice that we haven't asked the **if** statement to check the DISABLE property against a number. This is because where something can only either be TRUE (**1**) or FALSE (**0**), you don't need to specify the number. If you don't specify a number, it will automatically check if it is TRUE. If you use an exclamation mark (**!** – meaning **anything other than**), the **if** statement will instead check if the property is FALSE (see below for an example).

We will also add a line of code at this point that will have the effect of slowing down the movement of the Driftwood. We'll add this because this function will be used by the game approximately 36 or 37 times every second, and if we don't slow it down, the Driftwood would move far too fast! We'll slow it down by using the DELAY property of the item in the map. This is the code to do that:

```
if(!IsUpdate(ObjGet(idx,O_DELAY))) return;
```

This code gets the DELAY property of the item (**ObjGet(idx,O_DELAY)**), then uses a function called **IsUpdate** to check if the number in the DELAY property is **anything other than** the current frame number (by using **!** before the function name). If it is **anything other than** the DELAY property number, then the **return;** code will stop the function there.

The effect of this is that if the DELAY property is **3** for example (**3** is the default), then the game will run the function up to that point **3** times before the code allows it to continue. If the number is **8**, it will run it **8** times, etc. So the higher the number, the longer the game will have to wait before running the code below it. A DELAY value of **3** or **4** are usually fine for most purposes. Now that we've dealt with those two complicated lines, we'll move onto the meat of the function, which will actually move the Driftwood.

First, to make things easier, we'll define a couple of Object properties as variables:

```
y = ObjGet (idx,O_Y);  
bob = ObjGet (idx,O_TIMER);
```

The top line defines variable **y** as the 'Y position' property of Object **idx**. The bottom line then defines variable **bob** as the TIMER property of Object **idx**. You may notice in the map that there isn't an Object property called TIMER, but we'll define it later, so that the game recognises this new Object property.

Now we'll add the first part of the main movement code:

```
if (bob==3)  
{  
    ObjSet (idx,O_Y,y+2);  
    ObjSet (idx,O_TIMER,0);  
    ObjPresent (idx);  
    return;  
}
```

This checks if variable bob is equal to **3**. If it is, the code inside the **{** and **}** brackets is run. The first line sets the Y property of variable **idx** to '**y+2**'. The variable **y** is the Object's current Y position property, as set above. In this case, **2** is added to that property, effectively moving the Object down by **2**.

The second line sets the TIMER property of variable **idx** to **0**. This is what changes which piece of code is run each time, as you can see that this is what is checked each time (using variable **bob**). The third line is a simple line of code to ensure that variable **idx** (the Driftwood) is updated properly. If this code is missed out, sometimes the Object may not be shown in the game (especially when moving rooms). The fourth line of code simply stops the function there. This is to stop it carrying on and potentially clashing with code further down.

Now that part of the code has been explained, you can add the rest of the code, which is basically the above code duplicated 3 times:

```
if (bob==0)  
{  
    ObjSet (idx,O_Y,y-2);  
    ObjSet (idx,O_TIMER,1);  
    ObjPresent (idx);  
    return;  
}  
if (bob==1)  
{  
    ObjSet (idx,O_Y,y-1);  
    ObjSet (idx,O_TIMER,2);  
    ObjPresent (idx);  
    return;  
}  
if (bob==2)  
{  
    ObjSet (idx,O_Y,y+1);  
    ObjSet (idx,O_TIMER,3);  
    ObjPresent (idx);  
    return;  
}
```

You can see from this that all each of the 4 sections do is move the Object either up (by using minus -) or down (by using plus +) by a certain amount. They then change the value of Object property TIMER, which is then used to get the game to check a different section the next time it runs through the code.

The whole function should now look like this:

```
////////////////////////////////////
// Bob an Object up and down
////////////////////////////////////
func Move_Inwater(obj)
{
    idx = ObjFind(obj);

    if(ObjGet(idx,O_DISABLE)) return;
    if(!IsUpdate(ObjGet(idx,O_DELAY))) return;

    y = ObjGet(idx,O_Y);
    bob = ObjGet(idx,O_TIMER);

    if(bob==3)
    {
        ObjSet(idx,O_Y,y+2);
        ObjSet(idx,O_TIMER,0);
        ObjPresent(idx);
        return;
    }
    if(bob==0)
    {
        ObjSet(idx,O_Y,y-2);
        ObjSet(idx,O_TIMER,1);
        ObjPresent(idx);
        return;
    }
    if(bob==1)
    {
        ObjSet(idx,O_Y,y-1);
        ObjSet(idx,O_TIMER,2);
        ObjPresent(idx);
        return;
    }
    if(bob==2)
    {
        ObjSet(idx,O_Y,y+1);
        ObjSet(idx,O_TIMER,3);
        ObjPresent(idx);
        return;
    }
}
```

Don't forget we still have to define the new TIMER Object property! To do this, open up [gamedef.gs](#), and put the following code at the bottom of the page:

```
#def O_TIMER      40                // bobbing code timer
```

This defines Object property **40** as TIMER. Check in the properties of an Object in the map, and you will see that property **40** is described simply as 'USER'. All of these USER properties can be given names in the way described above, and used in the code.

Finally, we have to tell the game to use our new function! We do this using a function called **UpdateRoom_RX_RY()**, so as this will be used to move the Driftwood, start a new function at the bottom of the Driftwood code:

```
func UpdateRoom_2_2()  
{  
  
}
```

Here we replace **RX** (the Room X co-ordinates) with **2**, and **RY** (the Room Y co-ordinates) with **2**, as the Driftwood is in Room **2,2** in the map. Now in this function, all we need to do is add the name of our function, along with a semi-colon (as required on all code lines). Don't forget that the number in the brackets needs to be the number of the Object we want to move (the Driftwood):

```
Move_Inwater(2005);
```

This **UpdateRoom** function will be called 36 or 37 times a second, and will then run your new **Move_Inwater** function each time. Because of this, the Driftwood will now bob up and down in the water.

You may also have noticed that when the Driftwood moves, it will move into Room **3,2** as well, so you will need copy and paste the function you've just written, then modify it for Room **3,2** as follows:

```
func UpdateRoom_3_2()  
{  
    Move_Inwater(2005);  
}
```

If we hadn't created a new function for moving the Driftwood, we would have had to put the code from the **Move_Inwater** function into both of the **UpdateRoom** functions, which would have needlessly duplicated the code.

D. Use the Stick on the Driftwood

Now that we've done the puzzle for putting the Driftwood into the water, we can move onto the puzzle for getting it to move. This will be done by using the Stick item to prod the Driftwood, setting it moving back and forth.

Firstly of course, we need to place the Stick item in the map. So right-click and

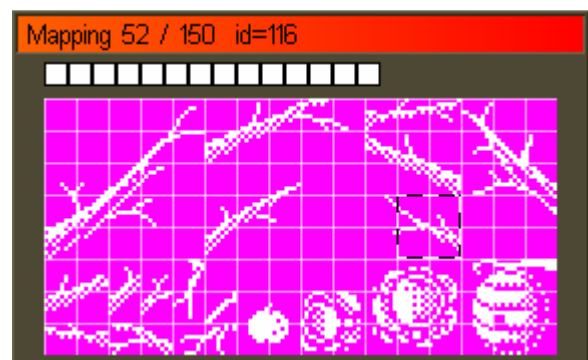


Fig. 13: Selecting the 'Branch' item

'pick brush' on the Pickaxe item (which isn't disabled to start with) to copy it, then go to the tile list, choose tile 116 (branches), and select the small branch shown in Fig 13.

Now keep the colour as Red, but change the ID to **1005**. Place it in the map on the left hand side of the right hand tree, as shown in Fig 14, and don't forget to set the Object property LAYER to '**1**'.

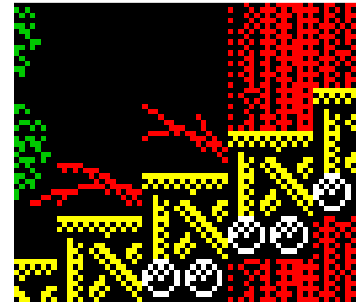


Fig. 14: Placing the 'Branch' item in the map

Now we have the item for the puzzle in place, and we already have both the Trigger and Driftwood for the puzzle in place. We would normally also put two small invisible Objects called 'waypoints' in the map to tell the game where to move the Driftwood to and from, however in this case, because we have the Driftwood bobbing up and down, adding waypoints would result in a rather strange movement. (This is because the code used with waypoints also considers the vertical position of the Object, as well as the horizontal position). We will therefore have to write a small piece of code ourselves, to get the Driftwood to move sideways.

We'll do this first, and then incorporate it into the existing code for the Driftwood Trigger.

So as before, create a new function by putting the following code below the **Move_Inwater(obj)** code in file **game.gs**:

```
////////////////////////////////////  
// Move an Object left and right  
////////////////////////////////////  
func Move_Leftright(obj,leftx,rightx)  
{  
  
}
```

This function will use 3 variables, as follows:

obj is the ID number of the object we're wanting to move (in this case, the Driftwood).

leftx is the furthest point that we want the object to move left.

rightx is the furthest point that we want the object to move right.

As this function is another one that will be called every 36 or 37 times, the first few lines will be the same as the code we wrote for bobbing the Driftwood up and down:

```
idx = ObjFind(obj);  
  
if(ObjGet(idx,O_DISABLE)) return;  
if(!IsUpdate(ObjGet(idx,O_DELAY))) return;
```

As with the 'bobbing' code, the top line finds variable **obj** in the map, and stores it as variable **idx**, to be used in the rest of the code. The second line stops the game going any further if the object is disabled, and the third line has the effect of reducing the amount of times the game runs through the code.

Next, similarly to the 'bobbing' code, we need to define the X position of the Object stored as variable **idx**. We'll store it, imaginatively, as variable **x**, using the following code:

```
x = ObjGet (idx,O_X);
```

Now that we have defined all the variables we need, we can start coding the main part of the function. This will be in two sections, one for moving left, and one for moving right. We'll distinguish between them by using the STATUS property of the Object which is stored as variable **idx** (the Driftwood), setting it as **1** if we want it moving left, and **2** if we want it moving right. The default for the STATUS property is always **0**, and it's easiest to use **0** for when it's not moving at all (before we start moving it).

So we first need to check the STATUS property of the Object stored as variable **idx**, and we do this with an **if** statement:

```
if (ObjGet (idx,O_STATUS)==1)
{
}
}
```

Now obviously we need to tell the game what to do if the STATUS property is **1**, so we'll start putting some code in between the **{** and **}** brackets:

```
x = x - 4;
```

This is simple. We're simply subtracting **4** from the value of variable **x**. This will move the Object left by **4**. Now we need the game to set the new position, and ensure that it is shown in the room. We'll do that with the following 3 lines of code:

```
ObjSet (idx,O_X,x);
ObjPresent (idx);
return;
```

This code should be fairly familiar from the 'bobbing' code above, but in essence, the first line sets the X position property of Object **idx** as variable **x**, the second line ensures that it is shown in the game, and the **return;** code on the third line will stop the function there, so that there is no conflict with the code that follows.

At this point, the code we've written so far will move the Object left indefinitely, without stopping. However we need it to stop at the furthest point left (stored

in variable **leftx**), so we'll use another **if** statement to check the new value of variable **x** against the **leftx** variable, to see if it's further left than the furthest point should be. So insert the following code directly after the **x = x - 4;** code, so that we can check variable **x** before we set the new position in the game (using the **ObjSet(idx,O_X,x);** code):

```
        if(x<=leftx)
        {

        }
```

This code checks if variable **x** is **less than** or **equal to** variable **leftx** (which is the furthest point left that we want to move it), and if it is, runs the code in the brackets.

The code in these brackets is two simple lines:

```
        x = leftx;
        ObjSet(idx,O_STATUS,2);
```

The first line sets variable **x** to be the same as variable **leftx** (in case it moved further left than **leftx**), and the second line changes the STATUS property of Object **idx** to **2**. This means that the next time the game runs the code, it will use the code for moving the Object right (determined by the STATUS property of **2**), instead of trying to move it further left.

At this point, you should have the following code:

```
////////////////////////////////////
// Move an Object left and right
////////////////////////////////////
func Move_Leftright(obj,leftx,rightx)
{
    idx = ObjFind(obj);

    if(ObjGet(idx,O_DISABLE)) return;
    if(!IsUpdate(ObjGet(idx,O_DELAY))) return;

    x = ObjGet(idx,O_X);

    if(ObjGet(idx,O_STATUS)==1)
    {
        x = x - 4;

        if(x<=leftx)
        {
            x = leftx;
            ObjSet(idx,O_STATUS,2);
        }
        ObjSet(idx,O_X,x);
        ObjPresent(idx);
        return;
    }
}
```

All that is needed to complete the code is to add in the code for moving the Object right, using the STATUS property of **2**. So copy the following code:

```
if (ObjGet (idx,O_STATUS)==1)
{
    x = x - 4;

    if (x<=leftx)
    {
        x = leftx;
        ObjSet (idx,O_STATUS,2);
    }
    ObjSet (idx,O_X,x);
    ObjPresent (idx);
    return;
}
```

And paste it into the function below itself, making the changes highlighted in Red below:

```
////////////////////////////////////
// Move an Object left and right
////////////////////////////////////
func Move_Leftright(obj,leftx,rightx)
{
    idx = ObjFind(obj);

    if (ObjGet (idx,O_DISABLE)) return;
    if (!IsUpdate (ObjGet (idx,O_DELAY))) return;

    x = ObjGet (idx,O_X);

    if (ObjGet (idx,O_STATUS)==1)
    {
        x = x - 4;

        if (x<=leftx)
        {
            x = leftx;
            ObjSet (idx,O_STATUS,2);
        }
        ObjSet (idx,O_X,x);
        ObjPresent (idx);
        return;
    }
    if (ObjGet (idx,O_STATUS)==2)
    {
        x = x + 4;

        if (x>=rightx)
        {
            x = rightx;
            ObjSet (idx,O_STATUS,1);
        }
        ObjSet (idx,O_X,x);
        ObjPresent (idx);
        return;
    }
}
```

Hopefully it is relatively easy to see now that the code to move the Object right is very similar to the code to move it left, with a few minor changes to take account both of the change in direction, and the comparison with variable **rightx**, rather than variable **leftx**.

Now we need to add this new function into the Room update functions that we coded earlier, so that the game will take it into account and move the Driftwood left and right.

So find the room update functions in the code, which should look like this:

```
func UpdateRoom_2_2()
{
    Move_Inwater(2005);
}
func UpdateRoom_3_2()
{
    Move_Inwater(2005);
}
```

All we need to do is add the name of our function to each of these room update functions. First though, we need to determine what the values of variables **leftx** and **rightx** are going to be.

To do this, we need to look at the map again. In the bottom left of the map, there are two numbers, which are the X and Y positions of the object you're currently holding. Change this object to the Driftwood by right-clicking on it and selecting 'pick brush'. This will enable you to use the Driftwood to accurately see what the furthest left and right X positions need to be. Move the Driftwood you're 'holding' over the water on the map until it's at a position which you feel is the furthest left it should go, then note the first number (the X position). For this example, the X position I've chosen is **608**. Now move the Driftwood right until it's as far right as you want it to go, and note the first number (the X position) again. In my case, it's **728**.

Now that we have these numbers, we can put the code into the room update functions, complete with all the variables we need (including the Object ID – **2005**), as shown in Red below:

```
func UpdateRoom_2_2()
{
    Move_Inwater(2005);
    Move_Leftright(2005,608,728);
}
func UpdateRoom_2_3()
{
    Move_Inwater(2005);
    Move_Leftright(2005,608,728);
}
```

All we need to do now is insert some new code into the Driftwood 'Trigger' code so that the player can use the Stick item to start the Driftwood moving.

So go back to the **UseObject_2004(idx)** function for the Driftwood, which should look like this:

```
func UseObject_2004( idx )
{
    if(ObjGet(idx,O_ID)==1004)
    {
        Message0(6,5,"YOU DROP THE DRIFTWOOD\nINTO THE WATER");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(2005);
        ObjSet(idx,O_DISABLE,0);
    }
    else
    {
        DropObject(idx);
    }
}
```

We need to add another **if** statement to check if the player is using the Stick, and if so, display a relevant message, and change the STATUS property of the Driftwood (Object ID **2005**) so that it starts moving. We can do this by copying the whole if statement from the above code, pasting it between the end of the **if** statement and the start of the **else** statement and modifying it as shown in Red below:

```
    if(ObjGet(idx,O_ID)==1005)
    {
        Message0(6,5,"YOU PROD THE\nDRIFTWOOD WITH\nTHE STICK");
        Message0(7,6,"AND IT STARTS\nTO MOVE!");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(2005);
        ObjSet(idx,O_STATUS,1);

        idx = ObjFind(2004);
        ObjSet(idx,O_DISABLE,1);
    }
```

Note that we've added an extra message line, and also added two lines to disable the Driftwood 'Trigger' (Object ID **2004**) now that it is no longer needed.

Don't forget to add the item name to function **ObjectsSetNames()**:

```
ObjSetName(ObjFind(1005),"A LARGE STICK");
```

There will however, be a couple of problems with this function now that we've introduced the above code into it.

Firstly, there is no **return;** code at the end of the **if** statement for the first item (**1004**). This isn't a problem when there is only one item being used, but if there are two, we have a problem. This is because if you use the Driftwood

item (1004), it will run that code fine. However because there is no **return;** code, the game will then also look at the **if** statement which checks if variable **idx** is 1005. Variable **idx** is currently set as Object 2005 (we just set it as that using the code **idx=ObjFind(2005);**), so as it isn't 1005, the game will then skip to the **else** statement at the bottom, which tells the game to Drop Object **idx**, which is currently Object 2005 (the Driftwood)! The effect of this is that after you use the Driftwood on the water, it will immediately appear next to you on the land (bobbing up and down too!), having been dropped there by the code. Have a go and try it if you want to, before we fix that problem by inserting a **return;** or two.

The second problem is that as the code stands, we can use the Stick item before the Driftwood is in the water! We'll fix this problem by again using and checking the STATUS property of the Driftwood, and changing it depending on whether the Driftwood has been used or not.

The lines of code to do this are shown in Red in the code below, and need to be added to your code as so:

```
func UseObject_2004( idx )
{
    drift = ObjFind(2005);
    if (ObjGet (idx,O_ID)==1004&&ObjGet (drift,O_STATUS)==0)
    {
        Message0 (6,5,"YOU DROP THE DRIFTWOOD\nINTO THE WATER");
        MessagePop ();
        InventorySub (idx);

        idx = ObjFind(2005);
        ObjSet (idx,O_DISABLE,0);
        ObjSet (idx,O_STATUS,5);

        return;
    }
    if (ObjGet (idx,O_ID)==1005&&ObjGet (drift,O_STATUS)==5)
    {
        Message0 (6,5,"YOU PROD THE DRIFTWOOD\nWITH THE STICK");
        Message0 (7,6,"AND IT STARTS\nTO MOVE!");
        MessagePop ();
        InventorySub (idx);

        idx = ObjFind(2005);
        ObjSet (idx,O_STATUS,1);

        idx = ObjFind(2004);
        ObjSet (idx,O_DISABLE,1);

        return;
    }
    else
    {
        DropObject (idx);
    }
}
```

As you can see, we've added 6 new bits of coding, two of which are **return;** code lines to stop the code clashing as described above. The first Red line of code defines variable **drift** as Object **2005** (the Driftwood). This is needed to check the STATUS property later on. We've added another check into the **if** statements (using the **&&** to signify an **and** statement), to check the STATUS property of variable **drift**, and we've added a line into the first item code (for **1004** – the Driftwood item), to change the STATUS property to **5** once that item has been used. You can see from the modified **if** statement for the Stick (item **1005**) that if the STATUS must be **5**, otherwise that code will not now run. This will stop the Stick item being used before the Driftwood item.

Now if you test the whole puzzle, you should be able to drop the Driftwood into the water, then prod it with the stick to start it moving. Congratulations, the player can now get to the other side of the water!

E. Use the Oil on the Switch to make the Lift move

Now we've written one puzzle to make something move, we're going to do another. This puzzle will involve the player trying to move a Switch, but being unable to, then using the Oil item on the Switch, which starts the Lift moving.

For this puzzle then, we'll need to place in the map the Oil item, a Switch, a Lift, and two waypoints for the Lift to use to move back and forth.

Firstly then, right click on the Pickaxe item and select 'pick brush'. Now go to the tiles list and into tile 211 (items1). Select a 16x16 box around the '3 in 1 Oil' tile, change the colour to Purple, the ID to **1002**, and place it down in the map on the ledge next to the Cheese item as shown in Fig 15. Don't forget to change the LAYER property to **'1'**.



Fig. 15: Placing the 'Oil' item in the map

Now we'll place the Switch item in the map. As this will have similar properties to the Rat, select the Rat by right-clicking on it and selecting 'pick brush'. Now go into tile 141 (mechanic1) and select the 16x16 square around the Switch in the bottom right hand corner. Change the colour of the Switch to Light Blue and the ID of it to **2006**. Also change the FLIP property to **0** (NONE), and the SHADER property to OPAQUE, as otherwise it blends in with the background. Place this down in the map near the Large Rock, as shown in Fig 16.

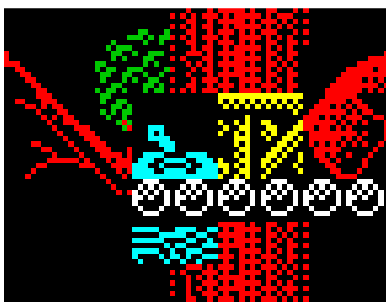


Fig. 16: Placing the Switch and Lift Objects in the map

After placing the Switch, we'll place the Lift in the map. We will have to be very careful where to place it, as the gap between the ledge and the rocks to the right is very small, and if we place it wrong, the player will be able to jump over the gap! If two platforms are at the same

height, Dizzy can jump over a gap of 8 blocks, so we will have to ensure that the gap is 9 blocks or more.

Right-click and 'pick brush' on the Driftwood Object in the water, then go into tile 125 (woods), and select a section of platform 16 wide x 8 high. I have used the platform in the bottom left of the tile. Change the colour of it to Light Blue, change the DISABLE property to NO, and the ID to **2007**. Now place it under the ledge as shown in Fig 16. We're placing it there because placing it anywhere between the ledge and the rocks will enable the player to jump over the gap.

Now we need to place two waypoints into the map. Select the purple 'Trigger' Object next to the Large Rock by right-clicking and selecting 'pick brush' as normal, then change the following properties of it:

ID: 2008
CLASS: WAYPOINT
TARGET: 2009

The TARGET property is the ID of the next Waypoint Object which the Lift will move to after it has moved to the position of this first Waypoint. You will also need to set USER property **32** to **3**. This is the speed that the lift will move at when moving to this waypoint.

Place it down in the map as an 8x8 block in the position shown on the left of Fig 17. Now place another one down in the position shown on the right of Fig 17, and reverse the ID and TARGET properties of it, so that the Waypoint on the right has an ID of **2009** and a TARGET of **2008**. Because they are targeting each other, the Lift will move back and forth between them once we set it moving. Speaking of which, we need to set the TARGET property for the Lift itself, otherwise it won't move anywhere! This TARGET property will only be used once, to get the Lift to the first Waypoint. So set the TARGET property of the Lift to **2008**.

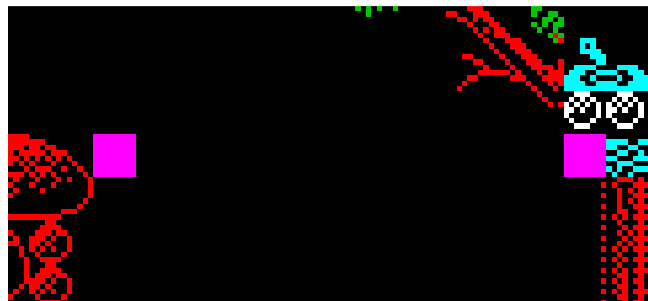


Fig. 17: Placing the Waypoints in the map

You may also notice the strange position of the right-hand Waypoint in the map. This is because Waypoints need to be set so that the top left corner of them are at the same point as you want the top left corner of the Lift to be at.

Now that we've placed everything in the map, we can start with the coding. Because we've used waypoints for the movement in this puzzle, the coding will be quite simple. First of all we need to code what happens when the player first actions the Switch, which will be very similar to what happens when the player first actions the Trigger for the Large Rock.

So find the first part of the code for the Large Rock Trigger, copy it, paste it below the Driftwood code, and change it as shown in Red below:

```
////////////////////////////////////
// Switch & Lift
////////////////////////////////////
func ActionObject_2006()
{
    idx = ObjFind(2006);
    if (ObjGet (idx,O_STATUS)==5)
    {
        idx = OpenDialogInventory();
        if (idx!=-1) UseObject (idx);
        return;
    }
    if (ObjGet (idx,O_STATUS)==0)
    {
        Message0 (4,4,"THE SWITCH IS\nSEIZED UP AND\nWON'T MOVE");
        MessagePop();
        ObjSet (idx,O_STATUS,5);
    }
}
```

Once again, you can see that very simple changes made to the code enable it to be used for a different puzzle. We'll do a similar thing with the next part of the code. The coding for the first Driftwood item is the most similar, so we'll copy and paste that function (**func UseObject_2004(idx)**), and remove the code for the second item. This will give us the code shown below:

```
func UseObject_2004( idx )
{
    drift = ObjFind(2005);
    if (ObjGet (idx,O_ID)==1004&&ObjGet (drift,O_STATUS)==0)
    {
        Message0 (6,5,"YOU DROP THE DRIFTWOOD\nINTO THE WATER");
        MessagePop();
        InventorySub (idx);

        idx = ObjFind(2005);
        ObjSet (idx,O_DISABLE,0);
        ObjSet (idx,O_STATUS,5);

        return;
    }
    else
    {
        DropObject (idx);
    }
}
```

Obviously we still need to change this, as we don't need the STATUS property checked for this puzzle, and we don't need anything disabling. We do however, need to FLIP the Switch Object once the Oil item has been used, and we also, of course, need to change some ID numbers too.

A few quick changes later, and we have the code shown below. Note the changes in Red, and also compare the two bits of code to see what has been removed:

```
func UseObject_2006( idx )
{
    if (ObjGet (idx,O_ID)==1002)
    {
        Message0 (5,4,"YOU USE THE\n3-IN-1 OIL ON\nTHE SWITCH");
        Message0 (6,5,"AND IT MOVES EASILY");
        MessagePop();
        InventorySub (idx);

        idx = ObjFind (2007);
        ObjSet (idx,O_STATUS,1);

        idx = ObjFind (2006);
        ObjSet (idx,O_FLIP,1);

        return;
    }
    else
    {
        DropObject (idx);
    }
}
```

Name the Oil in the **ObjectsSetNames()** function, as so:

```
ObjSetName (ObjFind (1002), "SOME 3-in-1 OIL");
```

And then we need to add an **UpdateRoom** function to move the Lift. So copy the **UpdateRoom_2_2()** function from the Driftwood code, paste it below the Lift code, and change the room numbers and code inside it to this:

```
func UpdateRoom_1_2()
{
    idx = ObjFind (2007);
    AIUpdateTrain (idx);
}
```

This Red code finds Object **2007** (the Lift) in the map, and then calls an AI function, which will move the Lift if the STATUS property of the Lift is set to **1** (which is what it is changed to in the **UseObject** function when the Oil item is used on the Switch Object).

The one last thing we need to do is ensure that the player can't get to the rocks from a higher ledge, so the end of the ledge above (with the hut on it) will have to be blocked off. Do this by right-clicking on the white platform and selecting 'select brush', then place a vertical wall (6 'blocks' high) at the left of the platform, as shown in Fig 18. By drawing it this high, we ensure that the player cannot jump on top of it, as the 5 'block' height above platform level is too high for Dizzy to jump onto.

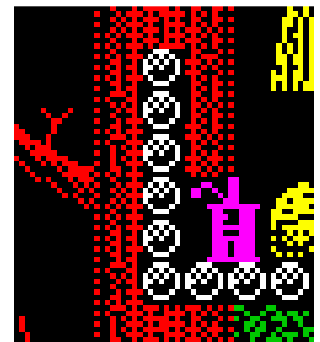


Fig. 18: Blocking off the higher ledge

F. Give the Garden Trowel to Dylan

This will be an even easier puzzle to code than the Lift puzzle, as it requires just three things added to the map, and two simple (by now) sections of code.

For this puzzle, the player will give a Garden Trowel to Dylan, and receive a Cupcake in return. We therefore need to place two items in the map – one enabled (the Garden Trowel), and one disabled (the Cupcake). We also need to place an Object (Dylan) in the map too.

So first of all, copy the Rat Object as normal. This Object has almost all the same properties as the Dylan Object will have, so by copying it, the only property we'll have to change is the FLIP property. Now select tile 175 (Dylan), change the colour to White, the FLIP property to **0** (NONE), and the ID to **2010**. Place it down in the map on the top right edge of the top ledge, as shown in Fig 19.

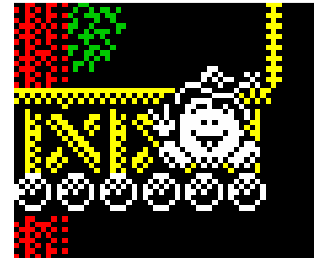


Fig. 19: Placing Dylan on the ledge in the map.

Now we'll place the items. Select the Pickaxe item as normal (this item is enabled in the map), and go into tile 214 (items4). There is a Trowel item in this tile, so select it, change it's colour to Green, and the ID to **1006**. Place it down in the map on the far side of the water as shown in Fig 20.



Fig. 20: The Garden Trowel item placed in the map.

Now for the Cupcake, you can use the Object you still have selected, so go into tile 212 (items2) and select the Cake item in the middle. Change the colour of it to White, the ID to **1007**, and the DISABLE property to YES. Now

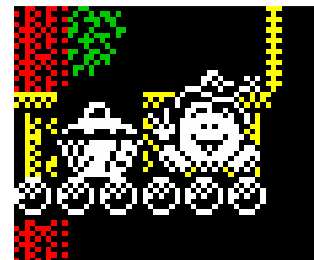


Fig. 21: Placing the Cupcake item in the map

place it down in the map next to Dylan, as shown in Fig 21. Don't forget to change the LAYER property of both the items to **'1'**!

Now we'll start coding the puzzle. We'll add the item names first, so that we don't forget later! So add the following two lines of code to the **ObjectsSetNames()** function, as usual:

```
ObjSetName(ObjFind(1006), "A GARDEN TROWEL");  
ObjSetName(ObjFind(1007), "A DELICIOUS CUPCAKE");
```

The first part of the code will be very similar to the Rat code, as we want Dizzy to talk to Dylan, so copy the **ActionObject_2002()** part of the Rat code, paste it below the Lift Switch code, and modify it as shown below in Red:

```

////////////////////////////////////
// Dylan
////////////////////////////////////
func ActionObject_2010()
{
    idx = ObjFind(2010);
    if(ObjGet(idx,O_STATUS)==5)
    {
        idx = OpenDialogInventory();
        if(idx!=-1) UseObject(idx);
        return;
    }
    if(ObjGet(idx,O_STATUS)==0)
    {
        Message1(4,4,"\"HELLO DYLAN,\nWHAT ARE YOU\nDOING HERE?\");
        Message2(6,8,"\"ADMIRING THE VIEW\nDUDE, IT'S AWESOME!\");
        Message2(7,9,"\"I CAN SEE ONE OF\nMY TOOLS FROM\nHERE TOO. CAN
YOU\nGET IT FOR ME?\");
        Message1(5,5,"\"SURE DYLAN, I'LL\nNOT BE LONG!\");
        MessagePop();
        ObjSet(idx,O_STATUS,5);
    }
}
}

```

This is the code for the initial conversation with Dylan. You can see the positioning of the message boxes changes so that they 'stagger' down the screen.

Next we need to add the code for what happens when you give Dylan the Garden Trowel. The easiest code to adapt for this is the code for the Lift Switch, so copy the **UseObject_2006(idx)** code for the Lift, paste it below the first section of the Dylan code, and modify it as shown below:

```

func UseObject_2010( idx )
{
    if(ObjGet(idx,O_ID)==1006)
    {
        Message0(5,6,"YOU GIVE THE\nGARDEN TROWEL\nTO DYLAN");
        Message2(3,3,"\"THANKS DUDE!\nHERE, HAVE MY\nLAST CUPCAKE\");
        Message1(6,8,"\"THANKS DYLAN!\");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(1007);
        ObjSet(idx,O_DISABLE,0);

        idx = ObjFind(2010);
        ObjSet(idx,O_CLASS,0);

        return;
    }
    else
    {
        DropObject(idx);
    }
}
}

```

All we're changing in the above code is the IDs used, the messages, and also ensuring that the Cupcake item (ID **1007**) is enabled, then changing the CLASS property of the Dylan Object (ID **2010**) to **0** (NONE), so that he can't be actioned again.

Make sure you run through the code above to ensure you understand how it works, and what the code is doing. Once you're satisfied with the above code and how it works, this puzzle is finished.

G. Give the Cupcake to Dora

This puzzle is almost exactly the same as the Dylan puzzle. The player will give the Cupcake item to Dora, and will get the Medicine item in return.

So firstly, select the Dylan Object in the map as normal, then select tile 173 (dora). Change the ID to **2011**, and place it down near the edge of the water, as shown in Fig 22.

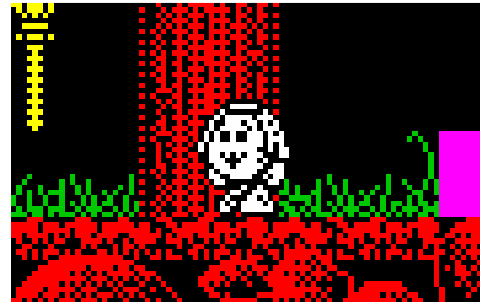


Fig. 22: Placing the Dora Object in the map near the edge of the water

As we have already placed the Cupcake item in the map, we only need to place the Medicine item in the map, so select the Cupcake item from the map as normal, then go into tile 211 (items1) and select the corked Medicine bottle item (top row, second from left). Change the ID to **1008**, and the colour to Purple. Place it in the map in front of Dora (Fig 23), setting the LAYER property to '**1**'.

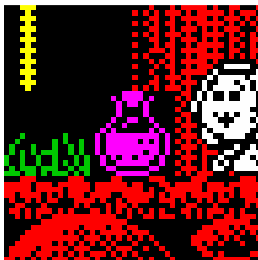


Fig. 23: Placing the Medicine item.

With this done, we can start with the coding. As with the Dylan puzzle, we'll start by naming the Medicine item in the **ObjectsSetName()** function:

```
ObjSetName (ObjFind(1008) , "SOME HORRIBLE MEDICINE" );
```

The good thing about the Dora puzzle being exactly the same as the Dylan puzzle is that we can use exactly the same code, and simply change the relevant ID numbers and messages that are shown.

So copy the entire Dylan code, and paste it in below the Dylan code. Now have a go at changing the relevant code before you look at the next page to see which code to change. Bear in mind the ID of the item to use (the Cupcake), is **1007**, the ID of the Object (Dora) is **2011**, and the ID of the item to get (the Medicine) is **1008**.

Once you've had a go at changing the code, have a look at the code below, with the changed code highlighted in Red, and compare it to the code you've changed to see if there is anything you've missed:

```

////////////////////////////////////
// Dora
////////////////////////////////////
func ActionObject_2011()
{
    idx = ObjFind(2011);
    if(ObjGet(idx,O_STATUS)==5)
    {
        idx = OpenDialogInventory();
        if(idx!=-1) UseObject(idx);
        return;
    }
    if(ObjGet(idx,O_STATUS)==0)
    {
        Message1(4,4,"\"HELLO DORA,\nWHAT ARE YOU\nHERE FOR?\"");
        Message2(6,8,"\"I'M TRYING TO FIND\nSOMETHING TO EAT\"");
        Message1(5,5,"\"I'LL FIND SOMETHING\nFOR YOU DORA\"");
        MessagePop();
        ObjSet(idx,O_STATUS,5);
    }
}
func UseObject_2011( idx )
{
    if(ObjGet(idx,O_ID)==1007)
    {
        Message0(5,6,"YOU GIVE THE\nCUPCAKE TO DORA");
        Message2(3,3,"\"OH THANK YOU\nDIZZY! CAN YOU\nGIVE THIS
MEDICINE\nTO GRAND DIZZY?\"");
        Message1(6,8,"\"SURE DORA.\n\"");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(1008);
        ObjSet(idx,O_DISABLE,0);

        idx = ObjFind(2011);
        ObjSet(idx,O_CLASS,0);

        return;
    }
    else
    {
        DropObject(idx);
    }
}

```

How does your code compare? Hopefully the only differences are in the messages. If so, you've just finished this puzzle!

H. Give the Ear Trumpet to Grand Dizzy

Again, this puzzle is almost exactly the same in execution as the Dora and Dylan puzzles. The player will give the Ear Trumpet item to Grand Dizzy, and will get the Driftwood item in return.

Hopefully by now you'll know what you need to do to code it, and what you need to put in the map, but I'll run through it just to be sure.

Firstly, you'll need to add a Grand Dizzy Object to the map, as well as an Ear Trumpet item. We've already added the Driftwood item to the map, so we don't have to worry about that. Select the Dora Object in the map as normal, then select tile 176 (grand). Change the ID to **2012**, and place it in the map in front of the hut, as shown in Fig 24.

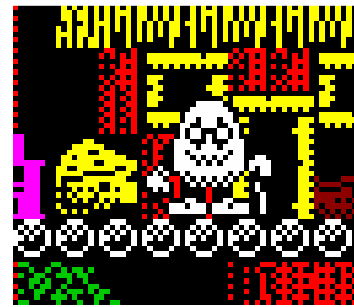


Fig. 24: Placing the Grand Dizzy Object in the map in front of the hut.

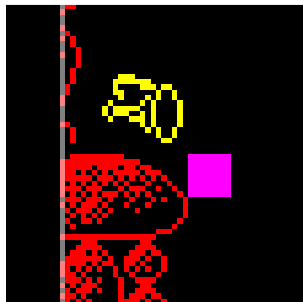


Fig. 25: Placing the Trumpet item in the map

Now select the Pickaxe item, and go into tile 215 (items5). Select the Trumpet item in the middle, change the colour to Yellow, and the ID to **1003**. Now place it down on top of the rocks on the far left of the playing area, as shown in Fig 25., and set the LAYER property to **'1'**.

Now we can start with the coding. Don't forget to name the Trumpet item in **ObjectsSetNames()**:

```
ObjSetName (ObjFind(1003), "AN EAR TRUMPET");
```

Now copy the entire Dylan code again, and paste it below the Dora code. Again have a go at changing the code to relate to the Grand Dizzy puzzle, remembering that the ID of the item to use (the Ear Trumpet), is **1003**, the ID of the Object (Grand Dizzy) is **2012**, and the ID of the item to get (the Driftwood) is **1004**.

Again, after you've had a go at changing the code, compare what you've got with the code below, with changes from the Dylan code highlighted in Red:

```
////////////////////////////////////
// Grand Dizzy
////////////////////////////////////
func ActionObject_2012()
{
    idx = ObjFind(2012);
    if (ObjGet (idx,O_STATUS)==5)
    {
        idx = OpenDialogInventory();
        if (idx!=-1) UseObject (idx);
        return;
    }
    if (ObjGet (idx,O_STATUS)==0)
    {
        Message1 (4,4,"\"HELLO GRAND DIZZY,\nCAN YOU HELP ME?\n");
        Message2 (6,8,"\"WHAT BOY? SPEAK\nUP, I CAN'T\nHEAR YOU!\n");
        Message1 (5,5,"\"I SAID, CAN\nYOU HELP ME?\n");
        Message2 (7,9,"\"OH! YES IT IS\nA VERY NICE\nDAY ISN'T IT?\n");
        Message1 (6,6,"\"I GIVE UP.\n");
        MessagePop ();
        ObjSet (idx,O_STATUS,5);
    }
}
```

The code above is for the first part, and the code below is for the second part of the puzzle:

```
func UseObject_2012( idx )
{
    if (ObjGet (idx,O_ID)==1003)
    {
        Message0 (5,6,"YOU STICK THE\nTRUMPET IN\nGRAND DIZZY'S EAR");
        Message0 (6,7,"AND OUT FALLS A\nLUMP OF DRIFTWOOD!");
        Message2 (3,3,"\"I SAY YOUNG DIZZY!\nI WAS USING THAT!\");
        Message2 (4,4,"\"MAKE YOURSELF\nUSEFUL, FETCH\nMY MEDICINE\");
        Message1 (6,8,"\"YES GRAND DIZZY\");
        Message0 (7,8,"YOU SCURRY AWAY\nWITH THE DRIFTWOOD");
        MessagePop ();
        InventorySub (idx);

        idx = ObjFind (1004);
        ObjSet (idx,O_DISABLE,0);

        idx = ObjFind (2012);
        ObjSet (idx,O_CLASS,0);

        return;
    }
    else
    {
        DropObject (idx);
    }
}
```

Again, if the only difference between your code and the code above is the messages, you've finished this puzzle!

I. Give the Medicine to Grand Dizzy

Now we come to the second Grand Dizzy puzzle. This will involve the player giving the Medicine item to Grand Dizzy, and getting the Cheese item in return. All the Objects and items needed have already been put into the map. They are:

- 1008** the Medicine item
- 1009** the Cheese item
- 2012** the Grand Dizzy Object.

For this puzzle, we'll slightly modify the Grand Dizzy code that we've just written, and we'll add an extra 'item' section to the second part of the code.

Because it isn't possible to get the Medicine item before giving the Ear Trumpet item to Grand Dizzy, we don't need to worry about checking the STATUS property to avoid the player solving the puzzles in the wrong order. Therefore it's just a simple matter to copy the Ear Trumpet item code (ID **1003**), paste it in below it, and modify it to suit the Medicine item code.

Have a go, then compare your code with the code below (changes highlighted in Red):

```
if (ObjGet (idx,O_ID)==1008)
{
    Message0 (5,6,"YOU GIVE THE MEDICINE\nTO GRAND DIZZY");
    Message0 (6,7,"AND HE DRINKS\nTHE WHOLE BOTTLE!");
    Message2 (3,3,"\"BURP!\");
    Message2 (4,4,"\"THANK YOU BOY.\nHERE, TAKE THIS\nLUMP OF
CHEESE\");
    Message1 (6,8,"\"THANKS GRAND DIZZY!\");
    MessagePop ();
    InventorySub (idx);

    idx = ObjFind (1009);
    ObjSet (idx,O_DISABLE,0);

    idx = ObjFind (2012);
    ObjSet (idx,O_CLASS,0);

    return;
}
```

You may think that this puzzle is now complete, but don't forget that we already changed the CLASS property of the Grand Dizzy object to 0 (NONE) after the player used the Ear Trumpet item. So if that piece of code stays, the player won't be able to action Grand Dizzy again to use the Medicine! So look again at the code for the Ear Trumpet (ID 1003), and **remove** the code highlighted in Red:

```
if (ObjGet (idx,O_ID)==1003)
{
    Message0 (5,6,"YOU STICK THE\nTRUMPET IN\nGRAND DIZZY'S EAR");
    Message0 (6,7,"AND OUT FALLS A\nLUMP OF DRIFTWOOD!");
    Message2 (3,3,"\"I SAY YOUNG DIZZY!\nI WAS USING THAT!\");
    Message2 (4,4,"\"MAKE YOURSELF\nUSEFUL, FETCH\nMY MEDICINE\");
    Message1 (6,8,"\"YES GRAND DIZZY\");
    Message0 (7,8,"YOU SCURRY AWAY\nWITH THE DRIFTWOOD");
    MessagePop ();
    InventorySub (idx);

    idx = ObjFind (1004);
    ObjSet (idx,O_DISABLE,0);

    idx = ObjFind (2012);
    ObjSet (idx,O_CLASS,0);

    return;
}
```

This will ensure that the player can still action Grand Dizzy, and use the Medicine on him. Once you've done this, the second part of the Grand Dizzy code should look like this:

```

func UseObject_2012( idx )
{
    if(ObjGet(idx,O_ID)==1003)
    {
        Message0(5,6,"YOU STICK THE\nTRUMPET IN\nGRAND DIZZY'S EAR");
        Message0(6,7,"AND OUT FALLS A\nLUMP OF DRIFTWOOD!");
        Message2(3,3,"\"I SAY YOUNG DIZZY!\nI WAS USING THAT!\");
        Message2(4,4,"\"MAKE YOURSELF\nUSEFUL, FETCH\nMY MEDICINE\");
        Message1(6,8,"\"YES GRAND DIZZY\");
        Message0(7,8,"YOU SCURRY AWAY\nWITH THE DRIFTWOOD");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(1004);
        ObjSet(idx,O_DISABLE,0);

        return;
    }
    if(ObjGet(idx,O_ID)==1008)
    {
        Message0(5,6,"YOU GIVE THE MEDICINE\nTO GRAND DIZZY");
        Message0(6,7,"AND HE DRINKS\nTHE WHOLE BOTTLE!");
        Message2(3,3,"\"BURP!\");
        Message2(4,4,"\"THANK YOU BOY.\nHERE, TAKE THIS\nLUMP OF
CHEESE\");
        Message1(6,8,"\"THANKS GRAND DIZZY!\");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(1009);
        ObjSet(idx,O_DISABLE,0);

        idx = ObjFind(2012);
        ObjSet(idx,O_CLASS,0);

        return;
    }
    else
    {
        DropObject(idx);
    }
}

```

Once you're happy that your code is correct, this puzzle is finished.

J. Use the Small Rock on the Button

The final puzzle will be to activate the Transporter. The player will do this by placing the Small Rock on a Button in front of the Rat. As an additional touch, we'll code it so that when Dizzy is stood on top of the Button, the Transporter is activated, but when he moves off it, it deactivates. Obviously when he places the Small Rock onto it, that will all become irrelevant.

For this puzzle we will need a Button 'Brush', a 'Trigger' Object for the Button (which will also serve as a 'Collider', to determine when Dizzy is stood on top of it), a Small Rock for scenic purposes (to display when Dizzy uses the Small Rock item), and a Beam in the Transporter.

First of all we'll place a Button in the map. This will be a Static Brush, rather than an Object, so select the Large Rock brush by right-clicking on it and selecting 'pick brush'. Go into tile 141 (mechanic1) and select the button near the bottom right corner. Change the colour to Purple and the ID to **2013**, then place it in the map half way between the rocks and the Rat, as shown in Fig 26.

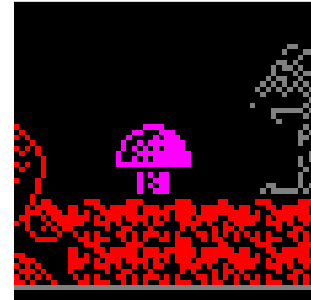


Fig. 26: Placing the Button Brush in the map.

Now select the purple 'Trigger' Object for the Driftwood in the usual way, change the ID to **2014**, and place it down (half height) on top of the Button as shown in Fig 27. Change the COLLIDER property of it to **1** (CALL

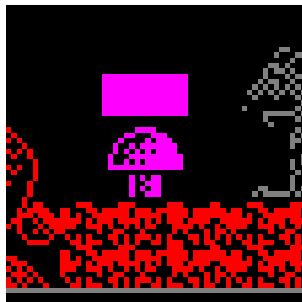


Fig. 27: The purple 'Trigger' on the Button.

HANDLER). This will enable us to write code that the game will run as soon as Dizzy collides with the Trigger, so that the Transporter Beam is activated when Dizzy stands on the button. The reason we haven't made it as big as the other Triggers is because we don't want Dizzy to 'collide' with it if he's jumping over it, and this will minimise the chances of that.

Ensure you have the properties of the Trigger set properly, because we are about to place the Small Rock

in front of it. Select the Small Rock item in the usual way, change the ID property to **2015** and the CLASS property to **0** (NONE). This will stop it being an item, and ensure it is just a bit of background scenery. Note also that the DISABLE property is **1** (YES). Place it in the map **in front of** the purple Trigger and the Button, as shown in Fig 28.

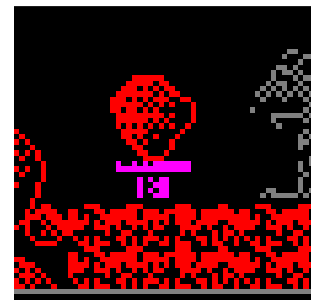


Fig. 28: The Small Rock in front of the Button.

Finally we need to place the Transporter Beam, so select the purple 'Trigger' Object for the Driftwood in the usual way, and go into the tile for it (tile **0** – default). Use the '**S**' key to turn Snap to Grid **off**, and select a rectangle 24 wide and 1 high. Now change the following properties of it:

ID: 2016
DISABLE: YES
CLASS: NONE
STATUS: 1

Now back in the map use the '**S**' key again to turn Snap to Grid **off** in the map, and place this rectangle down inside the Transporter, as shown in Fig 29. This will be the Transporter Beam. It doesn't need to be at exactly the same height, as it will move up and down in the game anyway.

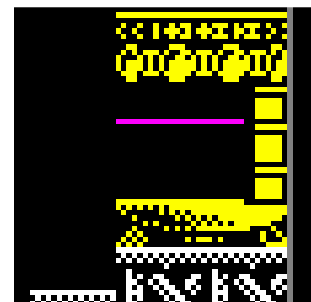


Fig. 29: The Transporter Beam in the map.

Now we can start coding this puzzle. The code for it will start off fairly similar to previous puzzles, as all it really boils down to is using the Small Rock item on the Button Trigger, enabling the Small Rock **Object** (not the item), and enabling the Transporter Beam Object. Once we've done this, we'll code the Beam so that it moves, and we'll code the Button to move (both when the Small Rock item is placed on it, and when Dizzy stands on it).

First things first then, copy the entire Large Rock code, and paste it in below the Grand Dizzy code. When changing the code to be relevant to the Button, remember that the Button Brush is ID **2013**, the Trigger Object is ID **2014**, the Small Rock Object is ID **2015**, and the Beam Object is ID **2016**.

The first thing to note is that because we'll be activating the Beam when Dizzy stands on the Button, there is no need for the **ActionObject** code. It should be obvious what needs to happen, so we don't need Dizzy to comment on it when he is standing on the Button. If there is no **ActionObject** function, the game will automatically run the **UseObject** function instead. The shortened code is shown below, with the changes from the Large Rock code highlighted in Red:

```
////////////////////////////////////
// Transporter Button
////////////////////////////////////
func UseObject_2014( idx )
{
    if (ObjGet (idx,O_ID)==1000)
    {
        Message0 (6,5,"YOU PLACE THE ROCK\nONTO THE BUTTON");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(2014);
        ObjSet (idx,O_DISABLE,1);

        idx = ObjFind(2015);
        ObjSet (idx,O_DISABLE,0);

        idx = ObjFind(2016);
        ObjSet (idx,O_DISABLE,0);
    }
    else
    {
        DropObject (idx);
    }
}
```

What the code above will do when the Small Rock item is used is display a message, remove the Small Rock item from the inventory, disable the purple Trigger Object, enable the Small Rock Object, and enable the Transporter Beam.

Next what we need to do is make the Beam move up and down when enabled, and enable it when Dizzy is stood on the Button.

To make the Beam move up and down, we'll copy and adapt the **Move_Leftright** function that we wrote earlier. So copy this function and paste it below at the bottom of the page. All we need to do now is modify it so that it uses the Y (Vertical) axis instead of the X (Horizontal) axis. The changes needed to do this are highlighted in Red in the code below:

```

////////////////////////////////////
// Move an Object up and down
////////////////////////////////////
func Move_Updown(obj,upy,downy)
{
    idx = ObjFind(obj);

    if(ObjGet(idx,O_DISABLE)) return;
    if(!IsUpdate(ObjGet(idx,O_DELAY))) return;

    y = ObjGet(idx,O_Y);

    if(ObjGet(idx,O_STATUS)==1)
    {
        y = y - 2;

        if(y<=upy)
        {
            y = upy;
            ObjSet(idx,O_STATUS,2);
        }
        ObjSet(idx,O_Y,y);
        ObjPresent(idx);
        return;
    }
    if(ObjGet(idx,O_STATUS)==2)
    {
        y = y + 2;

        if(y>=downy)
        {
            y = downy;
            ObjSet(idx,O_STATUS,1);
        }
        ObjSet(idx,O_Y,y);
        ObjPresent(idx);
        return;
    }
}

```

Next we need to add a line of code to the **UpdateRoom** function for this room in order to call the above code and move the Beam when the Beam's STATUS property is either **1** or **2**. So find the **UpdateRoom** function for Room **3,2** (in the Driftwood code), then find in the map the top and bottom Y positions that you want to move the Beam (Object **2016**) to, and add a line of code (shown below in Red) to the **UpdateRoom** function as follows:

```

func UpdateRoom_3_2()
{
    Move_Inwater(2005);
    Move_Leftright(2005,608,728);
    Move_Updown(2016,330,350);
}

```

Now we'll write the code that enables the Beam when Dizzy stands on top of the Button. As an extra little effect we'll make the Button move down when Dizzy is on top (As Dizzy has to be stood on top of the Button to place the Small Rock on it, the Button will already be down, so there will be no need to code it to move down when you use the Small Rock item on it).

This will require some new coding, although as you'll see, we can still copy bits of coding from elsewhere. Firstly though, write a new function below the Button code, as shown below:

```
func CollideObject_2014_1()  
{  
  
}
```

This creates a Collide function for Object ID **2014**, which will be called when Dizzy walks in front of Object **2014** (the Trigger) in the game (providing that Object has the COLLIDER property set to CALL HANDLER, and the DISABLE property set to NO). The number **1** in the function name is the Collider Mode. Mode **1** is 'when entering collision', mode **2** is 'while continuing to collide' and mode **0** is 'when exiting collision'.

Now copy the following code from the Button function **UseObject_2014(idx)**, and place it into the new function you've just written:

```
idx = ObjFind(2016);  
ObjSet (idx,O_DISABLE,0);
```

This will enable the Beam. To move the Button Brush down, copy the code below from the Rat function **UseObject_2002(idx)**:

```
idx = BrushFind(2003);  
BrushSet (idx,B_DRAW,0);  
GameCommand (CMD_REFRESH);
```

Paste it into your new function, and change it as follows (changes highlighted in Red):

```
idx = BrushFind(2013);  
BrushSet (idx,B_Y,382);  
BrushSet (idx,B_H,10);  
GameCommand (CMD_REFRESH);
```

This code finds Brush **2013** (the Button), changes the Y position property of it to **382** (6 lower than it is set to in the map), and the HEIGHT property (H) of it to **10** (6 less than it currently is, to remove the 'stand' below it). Note also that because we're changing a **brush**, not an **object**, we need to use **B_** rather than **O_** when we set it.

The last line is needed to refresh the brushes in the game after we've changed one or more of them (which we have just done). If this line isn't

included, the changes on the second and third lines won't show up in the game.

Once you've done this, the code should look like this:

```
func CollideObject_2014_1()
{
    idx = ObjFind(2016);
    ObjSet(idx,O_DISABLE,0);

    idx = BrushFind(2013);
    BrushSet(idx,B_Y,382);
    BrushSet(idx,B_H,10);
    GameCommand(CMD_REFRESH);
}
```

Now we've written the code to **enable** the Beam when Dizzy is stood on the Button, we also need to write code to **disable** it when he gets off it! This is however a fairly simple matter now that we've written the CollideObject function above, and we can just copy that function and set everything back as before. So copy that function, paste it below, and modify it as shown below in Red:

```
func CollideObject_2014_0()
{
    idx = ObjFind(2016);
    ObjSet(idx,O_DISABLE,1);

    idx = BrushFind(2013);
    BrushSet(idx,B_Y,376);
    BrushSet(idx,B_H,16);
    GameCommand(CMD_REFRESH);
}
```

We've changed the **Mode** part of the function title to **0** so that this function is only run 'when exiting collision' (when Dizzy stops standing in front of the Trigger). Other than that, we've disabled the Transporter Beam Object, and changed the Y position property and the HEIGHT property (H) back to what they were to start with.

With this function now complete, all the game puzzles have been completed. We will code the **ending** of the game later on in the tutorial.

Part 2 – Baddies and Coins

This part will show you how to place and code baddies, and how to place coins in the map and check for them in the code.

We'll add 4 baddies and dangers, and then add 5 coins to the map.

- A Adding and coding a Bat
- B Adding and coding a Spider
- C Adding Flames to the torches
- D Making Dizzy drown in Water
- E Adding Coins to the map
- F Hiding a Coin in the map

Part 2 is quite a small part, but will help you to add elements other than puzzles into your game.

A. Adding and coding a Bat

First of all we'll place a Bat baddie into the map. So select the Dora Object by right-clicking and choosing 'pick brush', then select tile 151 (bat) and change the ID to **3000**. Now place it down in the map in the position shown in Fig 30, and change the following properties:

COLLIDER: CALL HANDLER
CLASS: HURT
STATUS: 1
TARGET: 3001
DEATH: 1

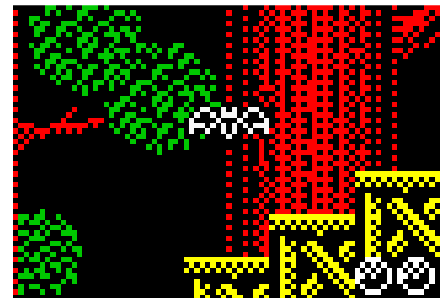


Fig. 30: Placing the Bat in the map

We're setting the STATUS property to **1** so that it will start moving straight away; and the DEATH property to **1** so that we can set some custom text if Dizzy is killed by the Bat.

Now we need waypoints for the Bat to move to, so select one of the Lift waypoints as usual, and change the ID property to **3001**, the TARGET

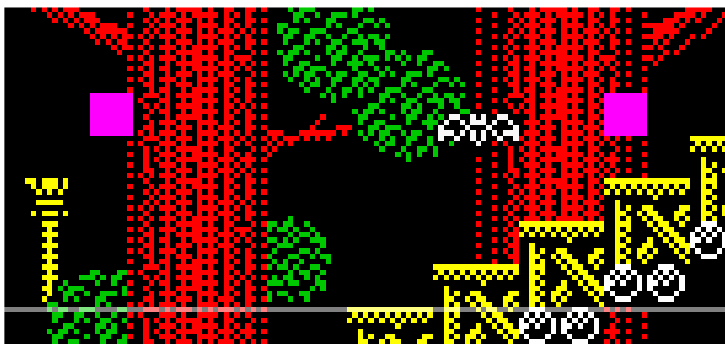


Fig. 31: The two Waypoints for the Bat

property to **3002**, and place it to the left of the Bat, near the yellow torch holder, as shown in Fig 31. Place another one to the right of the bat (Fig 31), with the ID and TARGET properties reversed, so that the ID is **3002** and the TARGET is **3001**.

The coding for moving the Bat is very simple, and virtually a duplicate of the Lift movement coding. Find the **UpdateRoom_1_2()** function in the Lift coding, and copy the whole function. Now paste it below the Transporter coding, and modify it as shown below in Red:

```
//////////////////////////
// Bat movement
//////////////////////////
func UpdateRoom_1_1()
{
    idx = ObjFind(3000);
    AIUpdateTrain(idx);
}
```

We've added the text at the top so that we can identify what this coding is for.

Now we'll add some code so that if Dizzy dies, there will be custom text telling the player that the Bat killed him. To do this, find the function called **PlayerDeathMessage(death)** in **game.gs**, which will look like this:

```
func PlayerDeathMessage( death )
{
    if(death== -1)                return "";
    // ...
    return "YOU HAVE DIED!";      // default
}
```

This code is used every time the player dies, to see if there is some custom text to display. If not, it will simply display **YOU HAVE DIED!** as a message.

To add custom text for the bat, we will add the following code (shown in Red):

```
func PlayerDeathMessage( death )
{
    if(death== -1)    return "";
    if(death== 1)     return "YOU WERE BITTEN BY\nTHE BAT AND DIED";

    return "YOU HAVE DIED!";      // default
}
```

This line of code checks if the **death** variable is set to **1**, and displays our custom text if it is. The **death** variable is set elsewhere in the scripts to be the DEATH property of the Object that killed Dizzy. In the case of the Bat, the DEATH property is set to **1**, so if Dizzy is killed by the Bat, our custom text will be displayed as a message.

It is worth noting at this point that you can only set custom DEATH properties for Objects. Setting them for Brushes won't work.

B. Adding and coding a Spider

Adding a Spider is very similar to adding a Bat, in that we'll use two Waypoints to move the Spider between. The difference here is that we'll also have a web that the spider moves up and down on.

Firstly select the Bat Object from the map as normal and change it to tile 164 (spider). Change the ID to **3003**, the TARGET property to **3005**, and the DEATH property to **2**. Place it into the map as shown in Fig 32 (you'll need to turn Snap to Grid **off** using 'S' to avoid cutting the edge of the spider off).

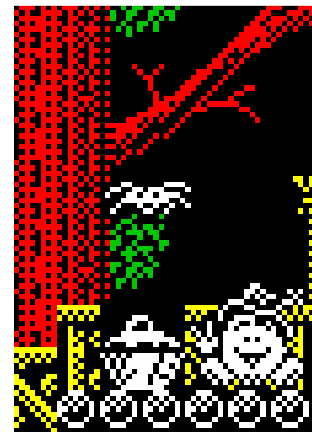


Fig. 32: Place the Spider above the Cupcake item

Next go into tile 112 (trees1) and select a 1x8 rectangle as shown in Fig 33. This will be the Web for the Spider.

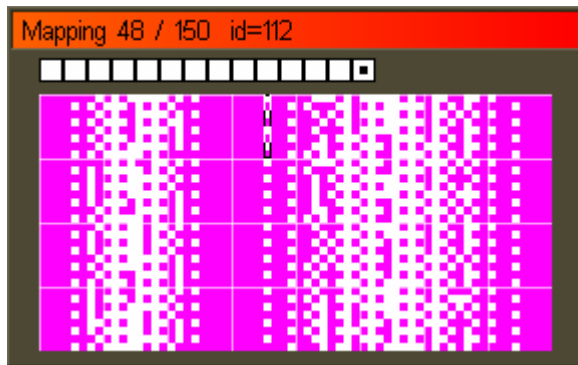


Fig. 33: Selecting the Spider Web

Now change the ID of the Web to **3004**, the TARGET property to **3003**, the COLLIDER property to **0** (NONE), the CLASS property to

0 (NONE), and the DEATH property to **0**. Place the Web into the map (turn Snap to Grid **off** to do this), running it from the Y shaped branch above the Spider, down to the Spider (Fig 34).



Fig. 34: Web in the map

Now turn Snap to Grid back on, and select one of the Waypoints from the Bat. Change the ID to **3005**, and the TARGET property to **3006**. Place it in the map near the top of the Web, aligned with the left-hand side of the spider. Place another Waypoint

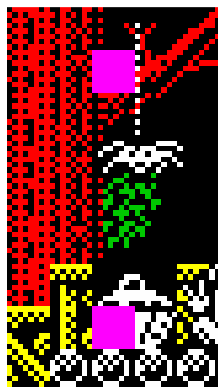


Fig. 35: Waypoints for the Spider

below the spider, with an ID of **3006** and a TARGET property of **3005**. These are both shown in Fig 35. Change the LAYER property of the Waypoint **3006** to **1**, otherwise it will be hidden behind the Cupcake item in the map.

While we've remembered about that, also change the LAYER property of both the Spider and the Web to **1**, so that they won't be hidden by the Cupcake when Dylan gives it to the player. To change the speed that the Spider moves up and down the Web, change the first USER property of Waypoint **3005** to **2** (to move slower up the Web), and change the first USER property of Waypoint **3006** to **4** (to move faster down the Web).

Now copy the coding for the Bat movement, and paste it in below it. Change it so that it is suitable for the Spider, as follows:

```

////////////////////////////////////////
// Spider movement
////////////////////////////////////////
func UpdateRoom_2_1()
{
    idx = ObjFind(3003);
    AIUpdateTrain(idx);

    idx = ObjFind(3004);
    AIUpdateChainLink(idx);
}

```

The line of code **AIUpdateChainLink(idx);** is to update the Spider Web.

Don't forget to also add another line of code to the **PlayerDeathMessage(death)** function, for the Spider:

```
if(death==2)      return "YOU WERE BITTEN BY\nTHE SPIDER AND DIED";
```

This concludes the Spider code.

C. Adding Flames to the torches

This section is very easy. Select the Spider Object as normal, and change the tile to tile 143 (flame1). Now change the following properties of the Flame:

ID: 0
STATUS: 0
TARGET: 0
DEATH: 3

Place the Flame down on top of each of the 5 torches in the map (You'll need to turn Snap to Grid **off** to get 3 of them lined up properly), as shown in fig 36. Now add the following line of code to the **PlayerDeathMessage(death)** function:

```
if(death==3)      return "YOU WERE BURNT BY\nTHE FLAME AND DIED";
```

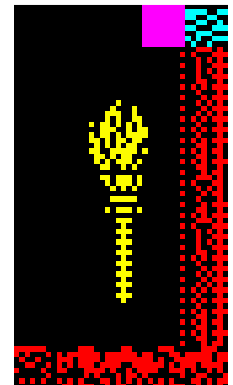


Fig. 36: A Flame in the map.

This shows the custom text as a message if the player dies from touching one of the Flames.

D. Making Dizzy drown in Water

This section will show you how to make Dizzy die if he falls into the water. There are two parts to this. Firstly actually killing him, and secondly making sure that he floats instead of falling through the water (as he currently does).

For the first part, right-click on the Water in the map, and select 'prop'. Now change the following properties of the Water:

COLLIDER: CALL HANDLER
CLASS: KILL
DEATH: 4

This will ensure that if Dizzy hits the water, he will be killed, and the **death** variable used for the custom death text will be set to **4**.

Next we need to ensure he floats rather than sinks once killed. To do this, select one of the Clouds using the normal method. The clouds are default Brushes, and is the easiest Brush to adapt for our purposes.

Change the tile to an 8x8 square of tile 0 (default), the colour to Light Blue, and then change the following properties:

MATERIAL: WATER
DRAW: MAT

Now draw a big block of this Brush below the waterline in the map, as shown in Fig 37. This will ensure that if Dizzy falls in the Water, he will float after being killed by the Water Object above.

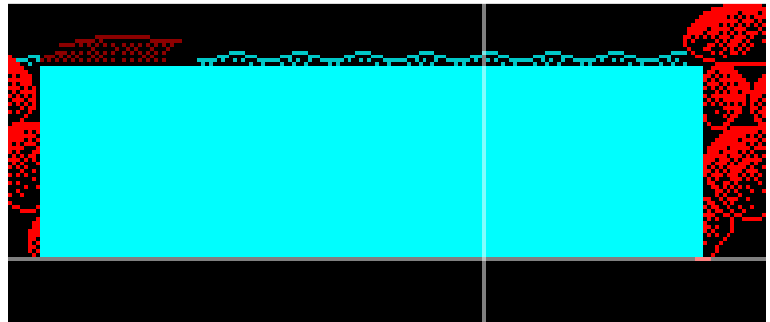


Fig. 37: The Water Brush placed in the map beneath the Water

Again, add the following line of code to the **PlayerDeathMessage(death)** function so that when he dies, there is a custom message:

```
if(death==4)      return "YOU FELL IN THE\nWATER AND DROWNED!";
```

E. Adding Coins to the map

This is one of the easiest parts of the whole Tutorial. Select the Ear Trumpet item, go into tile 210 (coins) and select the Coin. Now change the ID to **0**, and the CLASS property to **5** (COIN). All you need to do now is to place four of these in the map, as shown in Fig 38.

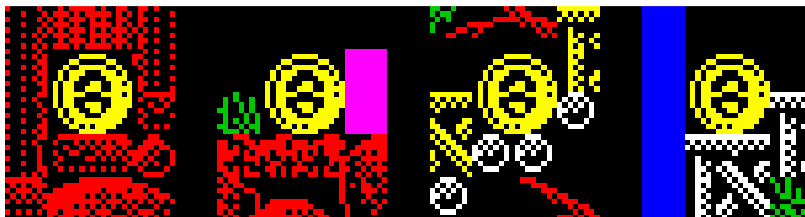


Fig. 38: The four coins in different places in the map

F. Hiding a coin in the map

This part will show you how to hide a Coin behind an item. Firstly find the railing below the right-hand waypoint for the Bat (Fig 39). Right-click on it, and choose option 'delete'. Now select a Coin in the normal way (if you're not holding one already), and place it down in the gap you've just created (Fig 40).

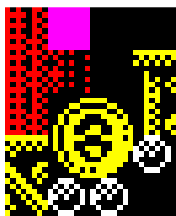


Fig. 40: Place a Coin here.

Now select the Ear Trumpet item in the usual way, then right-click on a railing and select 'pick tile'. This will change the Ear Trumpet item to be a Railing item. Now change the ID to **1020**, and place it down over the top of the Coin, so that it again looks the same as before (Fig 39). Don't forget to change the LAYER property of the Railing item to **'1'**, and add the following code to function **ObjectsSetNames()**:

```
ObjSetName(ObjFind(1020), "A PIECE OF RAILING");
```

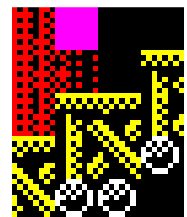


Fig. 39: Find this railing.

The last thing you need to do is change how many Coins the game thinks is in the map. Open up file **gamedef.gs** and find the code shown below:

```
#def MAXCOINS 4 // max number of coins or  
diamonds to find (set it to the number of coins you added in your map)
```

All you need to do here is change the number **4** to be the number of coins in the map (in our case, **5**).

Part 3 – Coding the Ending

To end the game, we'll code it so that Dizzy needs 5 Coins to activate the Transporter, but we'll also code it as a Collider, so that the player doesn't have to action it. If they have the correct number of Coins, Dizzy will move to the End Screen, a few message boxes will pop up, and the game will end.

First we'll place a Collider Object in the map at the right hand side of the Transporter. Select one of the purple Trigger Objects, and change the following properties:

ID: 2017
DISABLE: YES
COLLIDER: CALL HANDLER
CLASS: NONE

Now place it in the Transporter in the position shown in Fig 41.

Now we need to code it, so we'll write a new function for when the player collides with the Collider, and put it below the Transporter coding:

```
////////////////////////////////////  
// Transporter Collider  
////////////////////////////////////  
func CollideObject_2017_1()  
{  
  
}
```

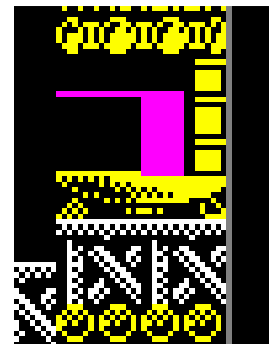


Fig. 41: The Collider in the Transporter

Inside this function we'll check the amount of Coins that the player has collected is the maximum number of coins:

```
coins = PlayerGet(P_COINS);  
if(coins==MAXCOINS)  
{  
  
}
```

If it is, we'll display a few messages, and call another new function (that we'll write shortly), to end the game:

```

    Message0(4,4,"YOU PUT THE COINS\nINTO THE SLOT");
    Message0(5,5,"AND THE MACHINE\nSTARTS TO WHIRR...");
    MessagePop();

    PlayerSet(P_COINS,0);

    End_Game();
    return;

```

Because we're 'using' the Coins, this code also sets the coin counter back to zero before calling the function that ends the game (**End_Game()**; - we'll write this next).

If the amount of coins isn't correct, we'll display a different message:

```

    else
    {
        Message0(4,4,"YOU SEE A SIGN\nON THE MACHINE");
        Message0(5,5,"TRANSPORTER -\nINSERT 5 COINS");
        Message1(6,8,"\"DARN IT I DON'T\nHAVE ENOUGH YET!\");
        MessagePop();

        return;
    }

```

This lets the player know how many coins they need to activate the Transporter. Once you've added this section of code, the whole function should look like this:

```

////////////////////////////////////
// Transporter Collider
////////////////////////////////////
func CollideObject_2017_1()
{
    coins = PlayerGet(P_COINS);
    if(coins==MAXCOINS)
    {
        Message0(4,4,"YOU PUT THE COINS\nINTO THE SLOT");
        Message0(5,5,"AND THE MACHINE\nSTARTS TO WHIRR...");
        MessagePop();

        PlayerSet(P_COINS,0);

        End_Game();
        return;
    }
    else
    {
        Message0(4,4,"YOU SEE A SIGN\nON THE MACHINE");
        Message0(5,5,"TRANSPORTER -\nINSERT 5 COINS");
        Message1(6,8,"\"DARN IT I DON'T\nHAVE ENOUGH YET!\");
        MessagePop();

        return;
    }
}

```

Before we move on to code the function we called **End_Game()**, we have to ensure that the Collider we've just coded is enabled once the Transporter is!

To do this, find the **UseObject_2014(idx)** function that enables the Transporter Beam once the Small Rock item has been placed on the Button, and add the following code shown in Red:

```
func UseObject_2014( idx )
{
    if (ObjGet (idx,O_ID)==1000)
    {
        Message0 (6,5,"YOU PLACE THE ROCK\nONTO THE BUTTON");
        MessagePop();
        InventorySub (idx);

        idx = ObjFind (2014);
        ObjSet (idx,O_DISABLE,1);

        idx = ObjFind (2015);
        ObjSet (idx,O_DISABLE,0);

        idx = ObjFind (2016);
        ObjSet (idx,O_DISABLE,0);

        idx = ObjFind (2017);
        ObjSet (idx,O_DISABLE,0);
    }
    else
    {
        DropObject (idx);
    }
}
```

This will enable the Collider (ID **2017**) once the Small Rock item has been placed on the Button. Now we can move on to coding the **End_Game()** function.

First of all we need to create a new function, so do this below the Collider code we've just written:

```
//////////////////////////
// Ending
//////////////////////////
func End_Game ()
{
}
}
```

Now we'll add code to it, based on what we need the game to do at the end. First of all, we need to ensure that Dizzy is facing forward, so we'll add this line (which will change the Dizzy tile to the default 'Dizzy idle' tile):

```
PlayerEnterIdle();
```

Then we need to ensure that the player can't move him, so we'll add this line to do that:

```
PlayerEnterScripted();
```

Once we've made sure he can't move and is facing forwards, we'll move him to the 'ending' screen, using the following line of code to place him at coordinates **120,374**:

```
PlayerSetPos(120,374);
```

Now we'll change the music, to give the ending a different feel:

```
MusicFade(0,1);  
MusicPlay(MUSIC_C);
```

The first line controls how the music fades in and out, and the second line determines which piece of music is played (MUSIC_C is defined in the **sound.gs** file as the music sample number **3** in the **data -> music** folder).

Next we'll add some good old messages:

```
Message1(3,3,"\"\"AHA, THIS MUST\\nBE MAGICLAND!\\\"");  
Message1(4,5,"\"\"I MADE IT!\\\"");  
MessagePop();
```

Finally, we'll add the most important part, the line of code that calls the ending menu:

```
OpenDialogFinish();
```

Put it all together, and you should have this:

```
////////////////////////////////////  
// Ending  
////////////////////////////////////  
func End_Game()  
{  
    PlayerEnterIdle();  
    PlayerEnterScripted();  
  
    PlayerSetPos(120,374);  
  
    MusicFade(0,1);  
    MusicPlay(MUSIC_C);  
  
    Message1(3,3,"\"\"AHA, THIS MUST\\nBE MAGICLAND!\\\"");  
    Message1(4,4,"\"\"I MADE IT!\\\"");  
    MessagePop();  
  
    OpenDialogFinish();  
}
```

Once you've done that, your game can now be played and completed, and is finished (well almost – see below)!

Part 4 – Additional bits

Now that your game is complete, you may think that's it, and you can release it. Not so fast! There are a couple of other minor things to attend to before you let the public loose on your masterpiece.

Firstly, you may notice that in the credits, the author is said to be a chap called **Alexandru Simion**! Obviously this needs to be changed, so open up **gamedef.gs**, and you'll see these two lines near the top:

```
#def GAME_AUTHOR      "ALEXANDRU SIMION"           // game's
author name (for the credits dialog)
#def GAME_TITLE       "DizzyAGE Default Template"  // game's
title (for the window title)
```

These two lines define the Author name (used in the credits menu), and the Name of the game (used in the window title – note that this IS case sensitive). Change these to your name, and the name you want to call your game. It is a good idea to put the version number in the title, so that if you have to update the game at some point (to fix bugs etc), people will know which version they're playing.

```
#def GAME_AUTHOR      "JAMIE DOUGLAS"             // game's
author name (for the credits dialog)
#def GAME_TITLE       "Make your own Dizzy Game v1.0" // game's
title (for the window title)
```

That will correct those little things. However if you press **F1** (for 'help') in the game, you will notice it still shows as Default Template v1.0 by **Alexandru Simion**!

To change this, you need to go into the **data** folder, and find the file called **dizzy**. Open this up and you'll find the following information on the top 3 lines:

```
game_title           = Default Template
game_version         = 1.0
game_author          = Alexandru Simion
```

Simply change the title and author as shown below in Red, and the **F1** information will change too.

```
game_title           = Make your own Dizzy Game
game_version         = 1.0
game_author          = Jamie Douglas
```

One last thing you may want to change is the message that pops up whenever you start a new game. To remove this, go into **game.gs** and find the function called **BeginNewGame()**. At the end of this function, you'll see the following three lines. Remove them and the opening message disappears.

```
// just a hello message
Message(14,6,"HELLO WORLD!",COLOR_MAGENTA,COLOR_GREEN);
MessagePop();
```

Part 5 (optional) – Change the HUD

The HUD (Heads Up Display) is the game border, including the section at the top where information such as the coin counter and energy bar is displayed.

In this part, we'll change the HUD of the game from the default one to a different one, and move the different elements of the HUD (lives, energy bar, coin counter, room title) to accommodate the new HUD.

The first part is the easiest – replacing the tile. The tile is called **1 menu.tga** and is found in the **data -> tiles -> menu** folder. You can alter this in most image programs (I use a free one called **GIMP**), but for this tutorial we'll simply replace it with the one I've included with this tutorial (the file called **New HUD.tga**). so delete the old file (**1 menu.tga** – Fig 42), then copy the new file (**New HUD.tga** – Fig 43) into the **data -> tiles -> menu** folder and rename it **1 menu.tga**. All tiles must



Fig. 42: The Default HUD



Fig. 43: The new HUD

have a number at the start of the name to identify them, and all must be in **.tga** format.

Before we go any further, I'll explain that the image shown inside the border is the image that is shown as a 'cover' when the game is loaded (after the 'loading' screen).

With that tile replaced, you'll instantly see a difference when you load the game.

However when you start playing the game, you'll see that all the elements displayed in the HUD are in the wrong place! This is what we will fix next.

Open up the script file called **handlers.gs** and scroll down to the function called **HandlerDrawHud()** near the bottom of the file. This function deals with everything that gets drawn on the screen, with the exception of the game window itself. In this instance, we're looking for 4 specific sections of this function. The ones tagged as **lifebar**, **credits**, **coins**, and **title**. We'll deal with them all in that order.

Lifebar, as you would expect, sets where on the screen the lifebar is positioned. To do this, it uses this code:

```
lifeid = 2; // life tile
w = 55*PlayerGet(P_LIFE)/100;
h = 6;
HudDrawTile( lifeid, 152, 5, w, h, 0, 0, w, h, 0, 0 );
```

The first line sets the ID of the tile to be used, in this case **2** (the lifebar tile). The second line works out how wide the tile is when shown on screen, then stores that value as variable **w**, and the third line sets the height of the tile and stores that value as variable **h**. It is the fourth line that we're really interested in, as it is this line that sets the positioning of the tile.

Before we start delving into the detail of this line of code, it is worth taking time to note the size of the default game screen. This is **256** pixels wide, and **192** pixels high. Of this, the window where the game rooms are shown is **240** pixels wide and **136** pixels high. The difference is an **8** pixel border to each side, plus a **40** pixel high 'HUD' at the top. The top left corner is therefore at X,Y co-ordinates **0,0**, and the bottom right is at X,Y co-ordinates **256,192**. Understanding this will help when it comes to working out where to put the different elements of the HUD.

Now I'll explain the **HudDrawTile(lifeid, 152, 5, w, h, 0, 0, w, h, 0, 0);** line of code. What it does is draw tile **lifeid** on the HUD, in an X position (from left to right) of **152** and a Y position (from top to bottom) of **5**, with a width *on screen* of variable **w** and a height *on screen* of variable **h**, a start X position *in the tile* of **0** (top left corner), a start Y position *in the tile* of **0** (top left corner), a width *in the tile* of variable **w** and a height *in the tile* of variable **h**. The second to last **0** sets the tiles FLIP property, and the last **0** sets the tiles FRAME property if it is an animated tile.

Hopefully from that you can see that the only two parts of the code that we need to change are the X position *on screen* of **152**, and the Y position *on screen* of **5**. Using trial and error, change these values until the lifebar is where you want it to be in the game. Note that you will have to close and re-open the game to see each change you make.

Once you've positioned it in the middle of the space for the lifebar, your line of code should look like this (changes in Red):

```
HudDrawTile( lifeid, 165, 9, w, h, 0, 0, w, h, 0, 0 );
```

Now we'll change the position of the **credits**. The code for this is slightly different, as it draws **text** on the screen, rather than a **tile**:

```
HudDrawText( fontid, 78+i*8,4,8,8, "@", 0 );
```

In this case, the line of code above draws text on the screen from font tile **fontid**, in an X position (from left to right) of **78+i*8** (this draws the first credit at X position **78**, then adds **8** pixels for each credit drawn, ensuring that all credits are drawn next to each other) and a Y position (from top to bottom) of **4**, with a width of **8** and a height of **8**. It uses the character "@" (which is an egg character in the font tile), and the **0** at the end is used to define the alignment – this is mainly used in messages and has no use here.

From this you can see that the only two parts of the code that we need to change are the **78** part of the X position of **78+i*8**, and the Y position of 4. Again use trial and error to change these until you get the credits in the position they should be in, which is as shown in Red below:

```
HudDrawText( fontid, 128+i*8,9,8,8, "@", 0 );
```

The line of code for the **coins** is exactly the same, except with different values as shown below:

```
HudDrawText( fontid, 56-w/2,4,w,8, text, 0 );
```

So use the same principles as above to move the coins into the correct place, as shown in the modified coding below:

```
HudDrawText( fontid, 108-w/2,9,w,8, text, 0 );
```

The last thing we need to move is the room **title**. This again uses the same code, but with different values:

```
HudDrawText( fontid, 128-w/2,29,w,8, RoomGetName(rx,ry), 0 );
```

All we need to do here is to move the text up slightly, so change the relevant part, and your code should look like the modified code below:

```
HudDrawText( fontid, 128-w/2,25,w,8, RoomGetName(rx,ry), 0 );
```

Once you've done this, you'll have a new HUD, and the elements displayed on it will be in the right places.

Part 6 (optional) – Animated Intro Screen

In this part, we'll create an animated intro screen, where Dizzy will jump around in a self-contained room, before the player starts the game.

Firstly we need to draw that room. The main thing to remember here is to ensure that Dizzy can't get out of the room. Select part of the ground by right-clicking it and choosing 'pick brush' as normal. Now in the blank room above the 'Weirdhenge' room, draw some ground as shown in Fig 44. Next go into tile 112 (trees1), and select the large tree. Now draw two trees, one at each side of the room. Note that as the Brush properties are still the same as the ground that you've drawn, the trees will Block Dizzy from leaving the room.

You can add more room decoration if you want to, but for the purposes of this tutorial, we'll leave the room like it is, and move onto the code.

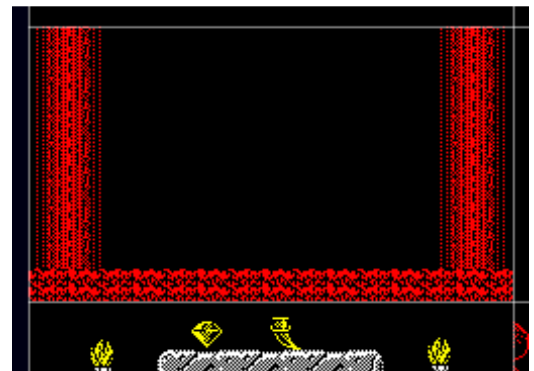


Fig. 44: Drawing the Intro Screen

There are three lines of code you need to change to get the game to use an animated intro screen. These are all in the **gamedef.gs** file. About halfway down the file you'll see three lines as follows:

```
#def MAINMENU_ATTRACT 0 // set to 1 if you want attract mode in
MainMenu (check mainmenu.gs)
#def PLAYER_MAINMENUX 400 // set the player's position x in the
mainmenu room (attract mode room)
#def PLAYER_MAINMENUY 246 // set the player's position y in the
mainmenu room (attract mode room)
```

The first line sets whether you want the 'attract' mode (animated intro screen) **on (1)** or **off (0)**. This is set to **off** as default. The second line sets the initial X position of Dizzy in the intro screen, and the third line sets the initial Y position of Dizzy in the intro screen.

All we need to do is change the 'attract' mode to **1 (on)**, and change the starting X and Y positions for Dizzy, as follows (changes highlighted in Red):

```
#def MAINMENU_ATTRACT 1 // set to 1 if you want attract mode in
MainMenu (check mainmenu.gs)
#def PLAYER_MAINMENUX 120 // set the player's position x in the
mainmenu room (attract mode room)
#def PLAYER_MAINMENUY 246 // set the player's position y in the
mainmenu room (attract mode room)
```

The position we've set him at in the code above is the middle of the screen we've just drawn. Happily, the Y position we need to set him at is the same as the default Y position, so we don't need to change that.

Now if you start the game, you will see after a while that the game view switches from the 'cover' to the animated intro screen. You may however notice a couple of things. Firstly it takes a while to switch to the intro screen, and secondly, the room name is '**...ATTRACT MODE...**'

If you want to change these, you'll need to open up file **menus.gs**, and look in the first function (**MainMenu()**) for the following code:

```
framemax = 474; // frames interval to toggle attract mode on and off
mode = 0; // 0=cover, 1=attract
```

The first line sets the amount of frames before the game switches from the cover to intro screen (and back again). Remember that there is approx 36 – 37 frames in a second, so we can see why it takes so long to switch! The second line sets whether the game shows the cover first (**0**) or the intro screen first (**1**). Remember that if you lower the value of variable **framemax**, you'll also lower the amount of time that the game shows the intro screen.

To change the room name when the game shows the intro screen, find the following code and change the part that says '**...ATTRACT MODE...**':

```
RoomSetName( GameGet(G_ROOMX), GameGet(G_ROOMY), (mode==0)?"PRESS
ACTION TO START":"...ATTRACT MODE..." );
```

The above code changes the room name depending on whether the game is showing the cover (**PRESS ACTION TO START**) or the intro screen (**...ATTRACT MODE...**).

Part 7 (optional) – Aqualung & Underwater

The last section we'll cover is how to add an Underwater area, where Dizzy can use an Aqualung to stay underwater.

We'll do this by adding a bottom to the lake, and slightly adjusting one of the puzzles in order to add the Aqualung.

Firstly go into the map, as we'll need to change a couple of things, and add the Aqualung. Right-click on the blue water tile (Fig 37, on page 49), and select 'send to back'. This will put the water tile behind the rocks in the map, allowing you to see the rocks. Next we'll select a rock in the normal way, and place it down at the bottom of the lake, to start to block off the bottom of the lake (so Dizzy doesn't fall through). Once you've done this a few times, you should have blocked the bottom of the lake off completely, and it should look roughly like Fig 45.

One thing we definitely need to do is ensure that jumping into the lake won't kill Dizzy! So we need to change the properties of the animated Water Object to be almost the same as the blue Water Brush. So change the following properties of the animated Water Object:

TYPE: STATIC
MATERIAL: WATER
COLLIDER: NONE
CLASS: NONE
DEATH: 0

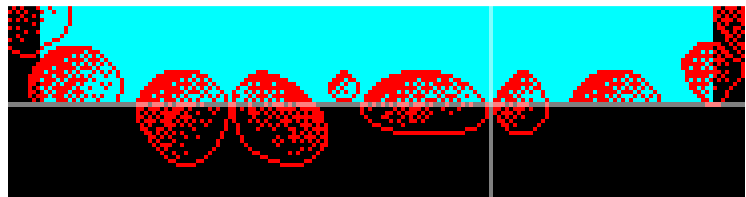


Fig. 45: Blocking off the bottom of the lake

Note that the only property that is now different to the blue Water Brush is the DRAW property. We have left it as IMG+MAT because we obviously need to see the animated water in the game! You will also notice that we have removed the value from the DEATH property. This is because we have changed the TYPE property to STATIC, and Static Brushes don't use the DEATH property. We have changed the TYPE property to STATIC because Dynamic Objects largely ignore the MATERIAL property.

Basically, because we need it to have the MATERIAL property as WATER, we need to have the TYPE property as STATIC. And because we need to have the TYPE property as STATIC, the DEATH property won't work for this, so we may as well remove it. This will cause a small issue later, as we'll see.

You can set scenery in the water too, however **always** either set the DRAW property of it to IMG only, or the MATERIAL property to WATER. Otherwise the game will see the scenery as 'air', and Dizzy will be able to breathe in front of it!

We now need to add an Aqualung item into the map. Because we want the player to **have** to use it, we need an incentive for the player to enter the water. We'll do this by changing the item Dora gives you, from the Medicine, to the Aqualung. We'll then move the Medicine item to the bottom of the lake, so that Dizzy has to use the Aqualung to get to the Medicine.

Firstly then we'll move the Medicine item. Right-click on it and select it as normal, then place it down at the end of the lake, as shown in Fig 46. Change the DISABLE property to **0** (NO), as we want it to be visible now, and change the LAYER property to **1**.

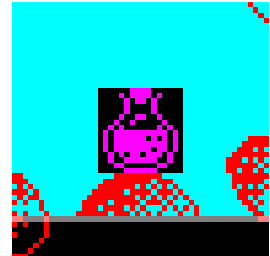


Fig. 46: Placing the Medicine in the Lake

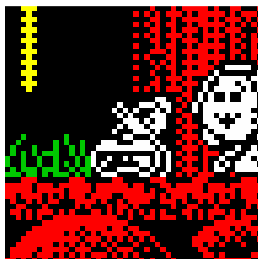


Fig. 47: Placing the Aqualung next to Dora

Now delete the old Medicine item next to Dora. Select the Cupcake item, go into tile 215 (items5), and select the Aqualung. Change the ID to **1010**, and place it down next to Dora, where the Medicine item used to be (Fig 47). Don't forget to change the LAYER property to **1**.

Once you have done this, we have finished with the map for now. We now have quite a few things to code. Firstly we'll change the Dora code and the messages between Dora and Dizzy to account for the different item the player will now get from Dora. Then we'll alter the code so that the water comes into play.

First of all however we need to add the Aqualung name to the bottom of function **ObjectsSetNames()**:

```
ObjSetName(ObjFind(1010), "DYLAN'S AQUALUNG");
```

Now find the **UseObject_2011(idx)** code in file game.gs:

```
func UseObject_2011( idx )
{
    if(ObjGet(idx,O_ID)==1007)
    {
        Message0(5,6,"YOU GIVE THE\nCUPCAKE TO DORA");
        Message2(3,3,"\"OH THANK YOU\nDIZZY! CAN YOU\nGIVE THIS
MEDICINE\nTO GRAND DIZZY?\");
        Message1(6,8,"\"SURE DORA.\");
        MessagePop();
        InventorySub(idx);

        idx = ObjFind(1008);
        ObjSet(idx,O_DISABLE,0);

        idx = ObjFind(2011);
        ObjSet(idx,O_CLASS,0);

        return;
    }
    else
    {
        DropObject(idx);
    }
}
```

This is the code for giving the Cupcake item to Dora, and receiving the Medicine from her. We need to change it to give Dizzy the Aqualung, and also to change the messages. We'll do this as follows (changes shown in Red):

```
func UseObject_2011( idx )
{
    if (ObjGet (idx,O_ID)==1007)
    {
        Message0 (5,6,"YOU GIVE THE\nCUPCAKE TO DORA");
        Message2 (3,3,"\"OH THANK YOU\nDIZZY! CAN YOU\nGIVE DYLAN HIS\n
AQUALUNG BACK?\");
        Message1 (6,8,"\"SURE DORA.\");
        MessagePop ();
        InventorySub (idx);

        idx = ObjFind (1010);
        ObjSet (idx,O_DISABLE,0);

        idx = ObjFind (2011);
        ObjSet (idx,O_CLASS,0);

        return;
    }
    else
    {
        DropObject (idx);
    }
}
```

Hopefully you can see from the above that not only have we changed the message, but we've also changed which item is enabled, so that the Aqualung now appears instead of the Medicine.

Now we need to set the water so that Dizzy will drown in it if he doesn't have the Aqualung. Go into file gamedef.gs and find the following code:

```
#def SUPPORT_WATERPLAY 0 // set to 1 if player can enter in
water, swim, etc, or to 0 otherwise
```

This code sets whether Dizzy moves differently in water or not. It is set to **1** if yes, and **0** if not. We want him to move differently, so change the **0** to **1**:

```
#def SUPPORT_WATERPLAY 1 // set to 1 if player can enter in
water, swim, etc, or to 0 otherwise
```

Scroll a bit further down until you come across the following line of code:

```
#def ID_SCUBA -1 // set scuba item's id if you have a
scuba in your game and you want to wear it as a costume for WaterPlay
```

This line of code is used to store the ID of a 'scuba' item (the Aqualung). It is currently set as **-1**, which tells the game that there isn't a 'scuba' item. We need to change it to the ID of the Aqualung (**1010**) as follows:

```
#def ID_SCUBA 1010 // set scuba item's id if you have a
scuba in your game and you want to wear it as a costume for WaterPlay
```

This will not only allow Dizzy to breathe underwater, it will also change how Dizzy looks when he is holding the Aqualung item.

Now test it, and it should all work perfectly. You may however notice three things. The first is that Dizzy can stay underwater without the Aqualung long enough to get the Medicine! The second is that the custom 'death' message no longer works, and the third is that there are no bubbles rising from him when underwater.

We'll deal with the first of these problems first. Go into file **def.gs**, and find this line of code near the top:

```
#def AIR_LEVEL          100          // player's air critical level
```

This is the amount of air that Dizzy starts with when he enters the water. Reduce it from **100** until you find a satisfactory level. **5** is low enough to stop Dizzy being able to carry the Medicine back (when he jumps into the water), but ideally we want to stop him even reaching it, as otherwise he can lose a life to get it back to dry land.

We could just set the CLASS of the Medicine to **0** (NONE) and then change it to **4** (ITEM) when Dora gives Dizzy the Aqualung, but you still wouldn't need the Aqualung to get it. So we'll do it a different way.

Open up file **player.gs**, and find the following code in function **PlayerUpdateWaterPlay()** near the bottom of the file:

```
if (PlayerGet (P_LIFE)>0)
{
    PlayerSet (P_LIFE,PlayerGet (P_LIFE)-2);
    if (PlayerGet (P_LIFE)<=0) PlayerSet (P_DEATH,0); // set cause of
death if needed
}
```

There are two things here. The first line inside the **{** and **}** tells the game how much energy to take away each time while Dizzy is drowning (**2**). This is what we'll change so that he is killed quicker.

The second line sets the DEATH property for the water if he drowns, so we'll change this too. Change the code so that it looks like the following (changes in Red):

```
if (PlayerGet (P_LIFE)>0)
{
    PlayerSet (P_LIFE,PlayerGet (P_LIFE)-10);
    if (PlayerGet (P_LIFE)<=0) PlayerSet (P_DEATH,4); // set cause of
death if needed
}
```

You should find that not only can Dizzy now not reach the Medicine item, but when he does die, the custom 'death' message will once again work.

Now we'll add some bubbles for when Dizzy is underwater. Select the Aqualung item in the normal way, then go into tile 238 (airbubbles) and select the largest bubble (turn Snap to Grid **off** to do this). Change the following properties of it:

ID: 4000
DRAW: IMG
DISABLE: YES
CLASS: NONE

Once you've done this, place it down in an empty room somewhere, where the player can't get to (turn Snap to Grid **off** to place all the bubbles, otherwise you'll cut the edges off). Once you've done this, place another 3 large bubbles down, with IDs of **4003**, **4006**, and **4009**. Now go back into the tile menu and select the medium size bubble. Place 4 of them down, with IDs of **4001**, **4004**, **4007**, and **4010**. Finally, go back into the tile menu and select the small bubble. Place 4 of these down in the map, with IDs of **4002**, **4005**, **4008**, and **4011**.

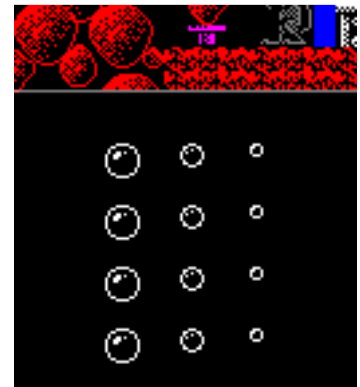


Fig. 48: Bubbles in the map

You now have 12 bubbles in the map (Fig 48), with IDs from **4000** to **4011**, and we now need to get the code to recognise them. We do this by going back into file **gamedef.gs**, and finding the following line of code:

```
#def ID_BUBBLES -1 // set this to the first valid  
bubble object. the rest of the PLAYER_BUBBLES bubbles have consecutive  
ids. used in WaterPlay
```

This line of code is used to store the ID of the first Bubble. The '**-1**' currently there tells the game that there are no bubbles in the map. Change this to the ID of the first Bubble, as shown below in Red:

```
#def ID_BUBBLES 4000 // set this to the first valid  
bubble object. the rest of the PLAYER_BUBBLES bubbles have consecutive  
ids. used in WaterPlay
```

Now when you start the game again, you'll find that Bubbles rise from Dizzy whenever he is in water. This ends the Underwater part.

Conclusion

Now that you've worked through the whole of this tutorial, you should have learned how to code the basic elements of Dizzy games. You should also now have your very own game, which you coded from scratch. I'm proud of you!

Hopefully you will also have found it easier than you were expecting. If you want to do more, there's nothing stopping you! Start drawing your own map, then start coding it – just like you have done in this tutorial. Refer back to it whenever you need to, and before long you'll have made a brand new Dizzy game, all on your own. Have fun!