GAMES THEME – Practical Module One B (P1B)

Pacman to Boulderdash

In this module we will look at three different styles of game: maze, platform and cell puzzle. A good deal of the work will be to do with methods of moving characters and monsters to make the game flow smoothly.

Sessions:

1. Pacman
2. Pacman extras
3. Platform games
4. Boulderdash
5. More on Boulderdash

Games – Pacman to Boulderdash – Session One
Pacman: Working with a Maze

Pacman was ... I was going to write some history here, but it is more complex than I remember. For a good summary try http://www.designboom.com/eng/education/pong.html instead. Here is what it says about Pacman.

In 1980, Namco game designer Moru Iwatani is tired of the glut of shoot-em-ups littering the arcades. He wants to create an arcade game that looks more like a cartoon than a video-game, and appeals to women as well as men. Like many famous figures, its origins have taken on mythical proportions. The designer was inspired by 'paku', a Japanese folk hero known for his appetite. His original design calls for an animated pizza with a missing wedge for a mouth running around a maze eating everything in sight. Technological restraints at the time, however, require a graphics scale-back to a simple, solid yellow circle. The large wedge of a mouth does remain, though, and the character and game is christened 'puckman', from the Japanese phrase pakupaku, meaning to flap one's mouth open and close. After the distinctive theme music plays, players find themselves guiding 'puckman' around a single maze eating dots, while avoiding the four ghosts 'blinky', 'pinky', 'inky' and 'clyde' (each with varying levels of hunting skills), who escape from a cage in the middle of the screen and will end our little yellow friend's life if they touch him. In each corner of the square playfield is a large dot that when eaten will turn the ghosts blue for a brief period, during which time the tables turn and 'puck' can eat the ghosts, leaving only the apparently indigestible eyes which make their way back to the cage for reincarnation into another ghost. During every screen a treat appears for the player under the ghost-cage, in the form of fruit or a bell or some other symbol waiting to be devoured. The game is deceptively simple, with only a four-position joystick needed to guide 'pac-man' around the maze, but with each successive screen the ghosts get faster and their time of blue-invulnerability less. Tension is added with a steady whining sound effect that increases in pitch as the game is played. The game is an absolute smash in Japan, following 'space invader's lead in causing another yen shortage nation-wide as tens of thousands of 'puckman' machines start gobbling them up.

From our viewpoint Pacman introduces the idea of a Maze game, in which a player explores a maze whilst avoiding opponents. We will build some of the game. Using only the graphic user interface of Game Maker it is straightforward to build the maze and have the Pacman move around. It is very tedious (although not impossible) to get the opponents to chase the Pacman in exactly the same way as they did in the original game, so we will do some simpler chasing manoeuvres.

At the end of this session, you should be able to…

- Create an animated sprite.
- Create a player and move it around a maze.
- Make opponents that move by themselves.

Getting Started

Files are provided with this module in directories on L: in the COMPX251 folder. The files for this session are in the Pacman directory. Save your game from time to time. Never trust computer software!

- Make a copy of the game files.
- Start Game Maker and create a new drag and drop project for Pacman.

Making the maze

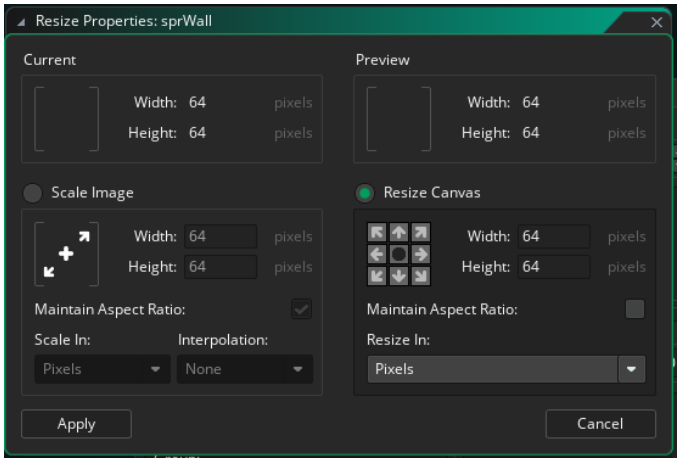
We will build the maze on a 32 pixel by 32 pixel grid.

- Open room0 and rename it to *Maze*
- Set the grid size to 32 by 32
  (Grid Options > GridX: 32 and GridY: 32)

Your room should be 640 by 480 pixels in size, displayed as a grid with 20 cols and 15 rows. The exact size is not important. The game will look better and have more room for a complex maze if you make it bigger, with a finer grid. However, the size suggested is better for experimental purposes as it makes it easy to see what is happening. Close the room.

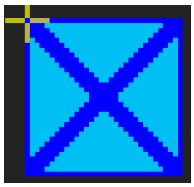
The next step is to insert the walls of the maze. We will eventually look at three ways of doing this, which allow successively more elaborate graphics. Because the way in which the player moves depends on the way in which the maze is built, we will just present the simplest method at this point, describe player movement, and then come back to maze construction later. The simplest way of building the maze is to create a *Wall* object that exactly fits a single grid square, in our case of size 32 by 32, and build up the maze from Walls. Rather than borrow a wall sprite, we will use the sprite editor to draw one.

Right click the Sprites folder in the Resources tab and select Create Sprite. Name it "sprWall". Click on the Resize button () and you will see this window:



Select 'Reise Canvas' and then you can change the width and height to 32 and then click Apply. You can click the magnifying glass icon to zoom in. Then click on Edit Sprite to go into the Sprite editor.

This is a simple editor, not unlike the Windows Paint program.



- Draw something that might make a wall segment. Mine looks like this. I used the line drawing tool with a medium thickness blue line, and pale blue flood fill for colouring the spaces. CTRL-Z will undo a single action, so if you work slowly you can correct mistakes and keep making progress.
- When you are happy with your wall, close the Image editor tab. This saves your image and takes you back to the Sprite properties window and then close that window.

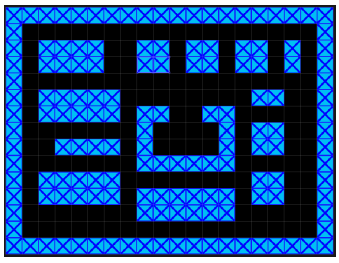
Now, back to making the maze.

- Create an object and name it "objWall".
- Set the object's sprite to be the sprite you created.

Note: The instructions will no longer tell you to do things like opening the objects property window. It is assumed that you just do them.

- Give your room a plain black background. You could do this by creating a background sprite and using it. For a plain coloured background it is easier not to have a background object as such, but just to choose a colour in the room's background layer property. (Click on the colour rectangle to open a colour selector.)
- You can now draw a maze using Wall objects. Try placing a few Walls in the room.

Note: if you are looking at the room in the editor, the grid lines of the room will affect the appearance. Try *running* the game to see exactly what it looks like.



- Finish drawing a maze with walls. Make sure that there is wall completely around the outside edge of the play area. You won't be able to draw as detailed a maze as the real Pacman game, because the 32 by 32 wall segments are too big. Here is our effort.

Hint: Try turning off the display of the grid while drawing the maze – it makes it easier to see the paths the Pacman can follow. To do this use the Toggle Grid button on the Menu Bar. 

Making the Pacman character

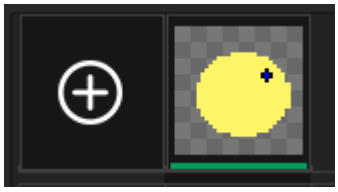
Fortunately the Pacman character is graphically simple – just a circle with a wedge missing. We can make ours more complex by adding eyes. If you feel more adventurous, you could try Ms Pacman from the 1982 game.

- Create a new sprite. Call it "sprPacman" (or "sprMsPacman"). Change the size to 32 x 32.
- Start the sprite editor.
- Draw a large yellow circle. Take care that it is centred and fills most (but not all) of the space. Use flood fill to fill it.

Notice that the area outside the circle is a grey white checkerboard. That represents the *transparent* area. Leave it unchanged and the Pacman will display in game as a circular shape against the room background.

- Draw a small circle for an eye. I used a black outline and blue interior.

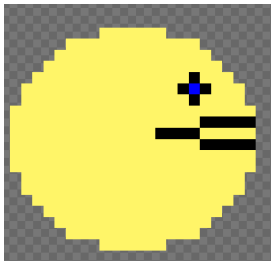
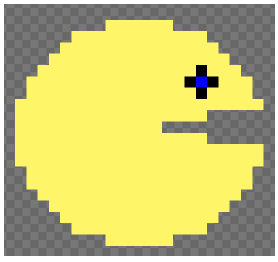
That will be the first frame of our animated sprite. In the preview bar up the top you will see a preview of the first frame.



We are going to make the Pacman sprite animated – it will open and close its mouth. Fortunately we are not required to draw each image separately. We can copy and modify images.

- Right click on the first frame and select Copy and then right click again and select Paste. This is an easy way to add frames based on the previous frame. If you just want a blank new frame then click on the '+' button.

- Select on the second frame so that you can edit it.
- Our objective here is to add a mouth, just a little open. To make the mouth we need to 'erase' pixels, leaving the open mouth gap transparent. There are various ways of doing this. Here is the one I used. My first step was to use the line tool to draw a straight black line for the top of the mouth and one for the bottom. Then I used the eraser to carefully remove the black pixels and the yellow between.

From  to 

- Repeat, to get two more images with the mouth more and more open.

Click on the Toggle Loop/PingPong button () once to set it to Loop mode. Then click the play button underneath to see a preview. Then click on the Toggle button again to set it into PingPong mode, it should look like this,  , then press the Play button again. Do you know what the difference is between those two modes? Leave it in PingPong mode as it looks better.

- Close the sprite editor tab and you should now have an animated sprite to use in the game.

Moving Pacman

The next step is to place Pacman into the maze and get it to respond to keyboard commands.

- Create a new object. Name it "objPacman"/ Set its sprite to be the Pacman sprite you created.
- Place a Pacman object into your room, in a position that allows it to move right.
- Try running the game. Your Pacman won't move yet, but it will be animated. You should be able to see its mouth opening and closing.

Sprite animation of this form is very helpful in games programming. It allows us to have things happening on the screen without having to specify the detailed movement and so forth. We can just concentrate on moving the Pacman as a whole. Movement in Pacman will be done with the keyboard. The real Pacman game had a joystick, wired to allow movement North, South, East or West. We will use the left arrow key for West, the right arrow key for East, etc. As discussed in the last module, there are many ways of allowing movement under keyboard control. In this case we will use a similar method to that used in the Space Invaders game – when the user presses a key we will start movement in the specified direction. One thing should be different though. In Space Invaders movement stopped when the user released the key. To be consistent with the original arcade game, in Pacman the character should keep moving after a keypress until it hits a wall. We can achieve that effect by not having a release event and having Pacman stop after a collision with the wall.

First – get Pacman to move.

- Add events to the Pacman object for Key Pressed <left>, <right>, <up> and <down>.
- Add a 'Set Direction Fixed' action to each Key Press event to move in the correct direction at speed of 4 (exactly – do not change this). Choose the appropriate direction for each key press.
- Test your game. You should be able to move the Pacman left, right, up and down. When you press a key the character should start moving. Movement should continue until you press another arrow key.

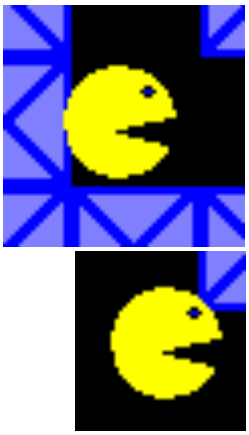
There is just one catch. We want to keep the Pacman moving in the passages of the maze. The first step is to program it to stop when it hits a wall.

- Add a collision event for Pacman – collision with a Wall. Set the speed to 0 to stop pacman moving.

- Test your program again. The Pacman should stay on the screen, but ...

A variety of nasty things may happen when you try to move the Pacman. Exactly what depends on some choices that you may or may not have made when setting up your sprites and objects. At this stage we will pause in the game development to do some systematic experiments. Your goal is to get a good understanding of the effect of each setting.
The critical settings are: For the Pacman sprite – whether it has *Precise collision checking* set; and for the Pacman and Wall objects – whether they have *Solid* set. (I am assuming that the wall sprite is not transparent, and that the Pacman sprite is transparent – if not, the game will have obvious graphical defects, so fix those settings and try again.)

The first experiment (results needed in session review)



Pacman sprite should have Collision Mask set to 'Precise' and Pacman and Wall objects should have Solid off (not checked). It is most likely that you have this already, as it results from accepting defaults. You will find that Pacman tends to get *stuck* in walls – notice the yellow pixels overlapping the wall.
What movement effects do you notice? What happens if we try to move Pacman up (North) from a position like that shown?

You will also find that it is possible to get Pacman into positions like this. How?

The second experiment (results needed in session review)

This time: Pacman sprite should still have Precise Collision Checking set on (no change) but both Pacman and Wall objects should have Solid **ON** (checked). Game Maker does collision checking differently for solid and non solid objects. The idea is that solid objects are solid and cannot therefore overlap (penetrate). If movement would lead to overlap, Game Maker has the movement back-off, so that the last step is not taken. Try again. You should find it impossible to get Pacman stuck in a wall as in the first image above. You will however find that something like that shown in the second image can still occur – thus. Why?

The third experiment (results needed in session review)

This time the Pacman sprite should have Collision Mask set to 'Rectangle'. Both Pacman and Wall objects should still have Solid on (no change). When Precise Collision Checking is on, a collision is only detected if a non-invisible part of the sprite touches something. When it is off, then the basic rectangle, regardless of transparency, is used for collision detection. In the case of the Pacman this is good. It leaves us with a nice buffer zone around the character that keeps it accurately in the centre of the passage ways. Try it. How easy is it to move Pacman into a side passage?

Leave the set-up in this configuration. This is the best one to use.

<u>Finalising Movement</u>

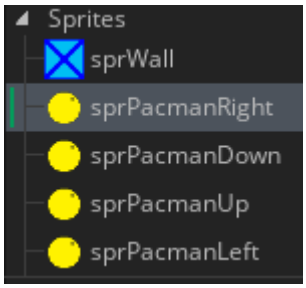
Most games cheat a little to help users with navigation. In first person 3D games it is rarely necessary to move accurately – the game will bounce you off the walls and floor to keep you moving forward. Usually the player doesn't notice. In our Pacman game as it stands, it is only possible to move into side passages if we time our key presses very precisely. The accuracy required is impractical for convenient game play.

There are some of possibilities for a solution. We could make the passages wider, or the ball narrower. To do this using the drag and drop actions available is not possible and you would need to write proper GML code to do this which is too advanced at this stage. What we can do to make it slightly better is that when a collision occurs we snap pacman back into the grid.

Open the pacman object and select the objWall collision event. Then add the 'Snap Position' action () from the Movement group and make sure the values are 32 x 32. Test your game, the movement is now better but still not great but it will suffice for this module. If you end up doing a maze game for your P2 this this level of movement would be sufficient unless you can figure out how to fix this on your own.

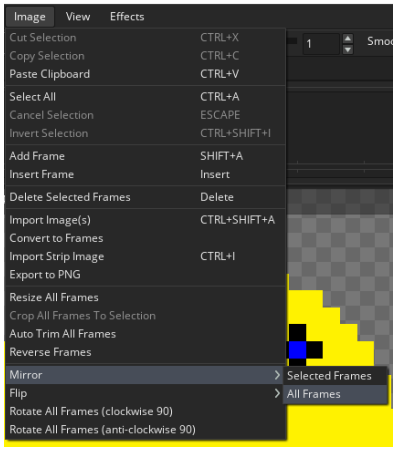
Making Pacman face the right way

There is one remaining problem with Pacman's movement. He always faces the same way. The only solution to this is to use a different sprite for each direction of travel. Fortunately the sprite editor makes it very easy to generate these. At the moment there is only one Pacman sprite. The first step is duplicate it, until we have four.



- Right click on the Pacman sprite (not object) and select *Duplicate* from the drop down menu. The sprite window will appear, give the new sprite the name "sprPacmanLeft" then click "OK".
- Repeat two more times using the names "sprPacmanUp" and "sprPacmanDown".
- Then rename "sprPacMan" to "sprPacmanRight".

Open up your pacman left sprite and then click on 'Edit Image'. From the Image menu select 'Mirror' and then select 'All Frames'.



- Mirror and Rotate as needed to get left, right, up and down.

It will not be necessary to make objects for each sprite. There is an action that allows an object to change its sprite. We can add that action to each of the keyboard events, so that the Pacman image turns along with its motion. We'll try two different ways of adding the 'change sprite action'. First, let's just add the sprite change to the keyboard event:

- Add a 'Set Sprite' action () from the Instances group to the <left> keyboard event to change the sprite to PacmanLeft.
- Make corresponding changes to each of the other key press events of the Pacman object.
- Try your program. It should look significantly better.

Dots

The maze is supposed to be full of dots for the Pacman to eat.

- Make a 32 by 32 sprite with a dot in the middle. Either make your own or use the one supplied.
- Make a dot object. Fill all empty spaces in the room with dots except for the cage.
- Load a sound (beeb) suitable for dot eating.
- Add an event to Pacman for colliding with a dot. (Dots need not be solid – why?) The actions for this event should be: destroy the instance of the dot; add to the score and play the (beeb) sound.
- You should also implement the scoring system using the instructions from the P1a.
- Test (I won't keep writing this – you should do it frequently)

Ghosts

In Pacman the opponents that players face are the Ghosts. Ghosts start in the cage at the centre of the screen and move around trying to catch the Pacman. In the real Pacman game the four ghosts had distinct 'personalities' – and used different strategies to pursue Pacman. Providing strategies for opponent pieces is a weakness in graphic game design systems. To provide a good strategy for a piece requires programming. In the Game Maker environment there is a built-in programming(scripting) language for this purpose and if you want sophisticated strategy there is no alternative but to learn how to use it. Writing that kind of script is beyond the scope of this game development introduction. However, there is a great deal we can do with simple strategies and many games don't require any strategy anyway. All games of the puzzle variety, most shooting games, for example, have no programmed strategy at all. The most complex game we will develop in this module is Boulderdash, and it has only minimal need for simple strategy.

At this stage the strategy we will create for the ghosts is just a random one. Their movement will be much like that of the Pacman, except that every time they reach a point aligned with the grid, they will choose their next direction at random.

- We have provided five ghost sprite images, called Inky, Blinky, Pinky, Clyde and Afraid. Create five sprites with appropriate names.

We don't need Afraid immediately – that is what the ghost becomes when being chased by an energised Pacman – but it will be needed later.

- Create a new object. Call it "objGhost".

There are going to be four ghosts, and they will all have exactly the same behaviour. To avoid duplicating the events and actions we will program just the Ghost object. Inky, Blinky, Pinky and Clyde will all use Ghost as a parent and thus inherit all the behaviour.

- Create four more objects. Call them objInky, objBlinky, objPinky and objClyde. Give them each their correct sprite. Make each of them have Ghost as their parent.
- Put an Inky, Blinky, Pinky and Clyde in your room. Usually they are placed in the 'cage' in the centre of the room. If you don't have a cage choose somewhere else to put them. One in each corner is an interesting possibility.

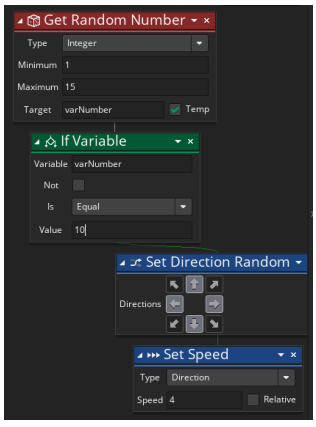
Events will be programmed for the Ghost object.

- Add a *step* event for objGhost. (Remember, *step* actions run on every time step of the game). The actions for step are: Move in a random direction The movement should be set to choose from N, S, E and W at random. Set the speed to 4. Test and see what happens.

The ghosts will move, but not stay inside the maze. To keep them in the maze we must add a collision event with walls. Having the ghost stop on hitting a wall will be enough. The next step event will try to start it moving again. It may move back into the wall, but the collision will stop it and eventually it should move away from the wall. Note that this is the same method we use for keeping Pacman in the grid, so we need non-precise collision detection and solid ghosts (however inappropriate that seems).

- Add a collision (with objWall) event to objGhost. Its action should be to stop moving and then add the Snap Position action and set the values to 32 x 32.
- Test again.

This is a workable solution to ghost movement, but it does give rather indecisive ghosts – that do a lot of stopping and starting. You can improve it a little, making it less likely that a ghost will stop and change its direction, by using a random number.



- Generate a random number between 1 and 15 (don't forget to give the target a variable name and tick the Temp option).
- Then use the If Variable action to test the variable is equal to 10. If it is true then choose a random direction and set the speed to 4.
- Try the program again.

This random ghost moving method isn't very good at getting ghosts out of large cages with small doors, but once they are out, it works quite well.

We haven't yet done anything about ghosts colliding with Pacman. That should be straightforward.

- Add a collision event (with objPacman) to the objGhost object. The action should be to play some sad music, and restart the game. We will work on handling game ending a little better later in the module. Of course there are lots of possibilities for improving the end. Playing a little animation of the Pacman's demise would be nice.

The Energiser

In the Pacman game there are four large dots (energisers). When Pacman eats one of these the ghosts turn blue for a period of time, and Pacman can hunt them. How can we make that happen? You might like to think of ways of doing it before reading my solution.
My solution. Although the ghosts change colour, it is Pacman that eats the dot and Pacman that changes its nature to become able to hunt ghosts. It seems therefore to make sense to change Pacman on eating the dot, rather than to change the ghosts.

- Create a new sprite called "sprEnergiser". Use the energiser image file, or draw your own. The one I have provided is a bit flat. It might be more interesting if you made it into an animated sprite with colours changing.
- Create an Energiser object. Put four of them into your room.

We now need to allow for Pacman colliding with an energiser. When this happens he must change to become the hunter Pacman? One of the actions in the Instances group is Change Instance (). This action changes an object into another kind of object. We could change Pacman into a wall, for example, although this would not be very useful. If we create a new object, HunterPacman, which is mostly just a copy of Pacman, but which will eat ghosts when it collides with them, then we have most of what we want.

One way of working would be to make a copy of Pacman, call it HunterPacman change the actions that needed to be changed. The better way of doing it is to use *parenting*. Pause for a moment here and think through the parenting we want. See if your view matches the following instructions.

- Save your game before this reorganisation of the program. Then, if something goes wrong, you can revert to the stored version.
- Rename the "objPacman" object to "objPacParent" and remove it's sprite.
- Create a new object, "objPacman", with a Pacman sprite (it doesn't matter which, why?). Make it solid and make its parent "objPacParent".
- The old Pacman object was used in two places in the game. One was in the Ghost object – it collides with Pacman. The other is in the game room. One Pacman object was placed into the game. You will find that both of these uses now refer to the PacParent (why?) and both must be changed to the new Pacman object.
- Create a new object "objPacHunter", with any Pacman sprite, solid, and with parent "objPacParent".
- Add an event to "objPacParent". On collision with an energiser, destroy the energiser, increase the score, and change instance to "objPacHunter". (Why add this to objPacParent and not objPacman here?)
- Test the program. You should find that Pacman seems to behave as normal, but that after eating an energiser it can run into Ghosts without being hurt.

The game now needs three things added to complete the interaction of Pacman and the ghosts. The first is that objPacHunter needs to revert to normal Pacman behaviour after a period of time. The second is that ghosts should turn blue and look frightened when Pacman is energised. The third is that objPacHunter should be able to destroy ghosts. Actually it doesn't destroy them. It just gets some points and sends them back to their cage.

Reverting to Pacman: One of the timing facilities in Game Maker is the *alarm clock*, introduced in session 2 of module 1. We can set it, and then after a period of time it gives an *Alarm* event. The idea here is to set an alarm when the objPacHunter is created (i.e.: Pacman becomes a PacHunter). Then when the alarm goes off, after say 5 seconds, we convert objPacHunter back to objPacman and all is as it was.

- Add a *Create* event to "objPacHunter".
- Add a Set Alarm Countdown action from the Instances group in the tool box. Set the number of steps to correspond to 5 seconds. Use in alarm no: Alarm 0.
- Add an *Alarm 0* event to "objPacHunter".
- Add an action to change the "objPacHunter" back into "objPacman" (change instance).

Ghosts Turning Blue: It is always better for objects to do things themselves, rather than have things done to them. So, if a ghost notices that Pacman has turned into PacHunter, it should turn blue, otherwise it should remain its normal colour. How can ghosts find out if Pacman is PacHunter – we can use the same method we used in Space Invaders to check to see if there were any invaders left on the screen.

- In "objGhost", open the *Step* event that already exists.

We don't have any way of knowing when Pacman will turn into PacHunter, so objGhost should just keep watching – at every step it should check. Because we are using the simple random movement strategy, the ghost doesn't actually change its behaviour when being hunted. All we need to do is change its sprite if there is a PacHunter on the loose after the if statement already there. The pseudo-code of what to add to objGhost's step event is:

```
Get the number of objPachunter objects in the room
IF number of objPachunter > 0
        Change sprite to afraid sprite
```

Add this code after the actions already in the Step event.

Of course the Ghosts must revert to the ordinary colours when there is no PacHunter around. We cannot do this in objGhost, as there is no way of knowing which is Inky, which is Blinky, etc. Therefore the individual ghosts must do their own check. Did you see this coming? Unfortunately we cannot just put *Step* events into each Ghost, because that would override and prevent the *Step* events in their parent Ghost from happening. There are two ways around this. One is to copy all the step actions into each of Inky, Blinky etc. That is really the correct way of doing it. The other is to cheat a little and that's what we will do now. Game

Maker actually provides three kinds of *Step* event: *Step, Begin Step,* and *End Step*. They differ slightly in their timing, but not in any way that matters here. We can just think of them as three different and independent *Step* events. (See online documentation for details.) We will add *Begin Step* events to each of Inky and friends.

- Add a *Begin Step* event to Inky.
- Add actions for the following pseudo-code:
- Get the number of objPachunter objects in the room
  IF the number of objPacHunters is equal to 0
    change the sprite to Inky
- Repeat the two steps for Blinky, Pinky and Clyde (hint: copy and paste the actions and then modify the change sprite action).

PacHunter can destroy ghosts: This is not new. We can add a collision (with PacHunder) event to Ghost. What should happen? The score will increase, the Ghost can jump back into the cage. (There are instructions in the third session of module 1 for finding the coordinates of a grid cell) If we wanted to send each Ghost to its own starting position, it would be necessary to work a little harder (how?).

- Add a *Collision* (with objPacHunter) event to objGhost.
- Add the actions: Add to score (but the Set Score action should apply to 'All' so that all ghost objects will add to the score); stop the Ghosts movement (why); jump Ghost to cage.

End of session

- Make sure that you save your game.

Session 1: REVIEW PAGE **Name: _____**

Questions: Complete the following questions and be prepared to demonstrate techniques to your demonstrator. (You can then have your work verified.)

1. Explain the difference between Loop mode and PingPong mode when previewing an animated sprite.

   Loop Mode - circle through the animation
   Ping Pong mode - 1st to last then last to 1st

2. Why do we use sprite animation – why not just use events and have a series of simple single image sprites?

   While it can be done just using event and a series of simple
   single image sprites but sprite animation is standard and powerful tool in game development for creating visually appealing and engaging gaming experiences and it is more efficient.

3. Experiment one: What does happen if we try to move up from the position given? Why? How do you get Pacman into the second difficulty show.

   If Pacman gets stuck in a wall, trying to move Pacman up (North) from that position will result in Pacman not moving. Because Pacman got stuck in the wall.
   Both Pacman and Wall objects should have Solid ON

4. Experiment two: How did Pacman get into this position? Why did it happen?

   By making Both Pacman and wall sloid they will not collide with each other.
   GameMaker ensures that these objects cannot overlap or penetrate each other. If movement would lead to overlap, GameMaker prevents it by moving the object back to its previous position.

5. Experiment three: How easy is it to turn into a side passage? Why?

It is not perfect but i is a lot easier to turn into a side passage because it create a nice buffer zone around the character that keeps it accurately in the centre of the passage ways.

6. When setting the score in the obhGhost object why did you need to make it apply to 'All' instead of 'Self'?.

Because we can keep the score for all ghost objects that will add to the score

Verification: To assess your competency of this material your demonstrator will verify that you have completed the above questions, and can do the following:

1) Add a super power up object with a suitable sprite which will destroy all ghosts when any type of pacman eats it. It also adds 1000 to the player's score.

Games – Pacman to Boulderdash – Session Two
Pacman: Finishing Touches

MORE AND MORE AND MORE AND MORE AND MORE AND MORE

A feature of commercial computer games is often the amount of detail that has been programmed in. Even in the original Pacman there is more detail. Ghosts return to the cage as eyes only. Just before the energiser effect runs out the frightened ghosts change appearance slightly. A good games programmer looks for extra ideas to entertain the users of the game. A great deal of test playing will be done, to get the difficulty level just right – not too easy, but not impossible either. At this stage we have the basic functionality of Pacman. A lot more can be done.

Creating your own High Score Table

Add a new room and make the size the same size as your maze and change the name to HighScoreTable. This room will be used to display the high score table but you will need to write some code to do this.

Game Maker will store the high scores for you but you need to retrieve the high scores and display them.

Open the "objGhost" object and select the collision event with "objPacman". Delete the restart room action and then add the "Execute Code" action () from the "Common" group. You will see a code editor box appear, you may need to resize things to see it properly. You can now type in GML code in this window which will be executed by the game. You will need to get the players name and then add their name and score to the high score table in the game. You will see the code editor appear and type in the following (I have added comments to explain what the code is doing which you should type in as well):

```
//Create a variable to store the name of the player
var name = "";
```

This code statement creates a variable which can store a value for you. It is called "name" and we set it to an empty string when we create it so that it is empty. Now we will get the player to enter their name.

```
//Ask the player to enter their name and store it
name = get_string("Please enter your name: ", "No Name");
```

In the get string command the first part is the prompt to the user and the second part is the default value if the user does not enter a string.

```
//Add the name and score to the high score table.
//The method will only add the name and score if the
//score is high enough to get on the table.  Use the name
//of the global variable you created when implementing the
//scoring system
highscore_add(name, global.playerScore);
```

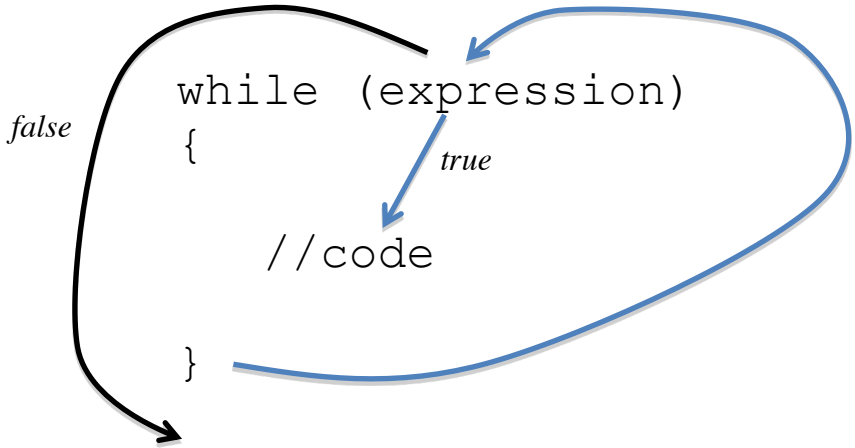
Add an "Exit Game" action from the "Game" group in the toolbox after the Execute code action otherwise Pacman keeps colliding with the ghost and it will keep asking you for a name. You should run the game and test that it asks for your name when you die.

Create a new font called fntHighScore for the high score table, make the size reasonably large but not too large (something between 14 and 18 is good).

We will need to create an object for displaying the high score table as the code needs to be in a Draw Gui event. Create an object called "objHighScoreTable" with no sprite. Add a "Draw Gui" event to the object and add an "Execute code" action to the event.

We want to display only the first 10 high scores. We could write the code to display a high score and copy and paste that code 10 times. Do you think this is a good way to write the code?

You are correct, this is not a good way to write the code. A better way is to repeat the code many times in a repetition structure. We will be using a while loop.



If the expression is true then the code inside the loop is executed. At the bottom of the loop it then goes back to the expression and repeats again. When the expression becomes false it stops executing the code inside the loop and carries on with any statements after the loop.

We use relational operators to write an expression which evaluates to true or false. These operators are:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Here is an example which loops from 1 to 5 (i.e. loops 5 times). "i" is a variable used to count the number of times the loop has repeated, variables used in this way are commonly called counter variables.

```
var i = 1;
while (i <= 5)
{
     //code
     i++;  //Adds 1 to the variable counter variable
}
```

Do you know why we must add 1 to the "i" variable before the loop ends?

The pseudo-code for displaying the high score table is as follows:

```
Declare variables
Set the font
Set the colour of the text
WHILE high score position <= 10
     GET the name for the current high score position
     Get the score for the current high score position
     Get the height of the string using the current font
     Draw the name at the current x and y position
     Draw the score at the current x position + a gap amount
        and at the current y position
     Shift the y position down by height of the   string
     Add 1 to the current high score position
ENDWHILE
```

Now in the code window we can translate the pseudo-code into actual game maker code.

We first need variables to store various values. I have used comments to explain what the variables are for. You should add the comments to your code as well to remind yourself what you have done.

```
//Counts the number of high scores
var i = 1;
//The name of the player that got the current high score
(default to No Name)
var name = "No Name";
//The score value of the current high score
var playerScore = 0;
//The x position to display the current high score
var xpos = 50;
//The y position to display the current high score
var ypos = 0;
//The height of the string to display
var fontHeight = 0;
//The gap between the name and the score
```

```
var gap = 150;

//Set the font to be used when drawing the names and scores
draw_set_font(fntHighScore);
//Set the colour for the text (try other colours later)
draw_set_colour(c_orange);
//Get the height of the font
fontHeight = string_height(name);
```

Now write a while loop with a condition that will loop 10 times. Get the instructor to check your loop before continuing.

The first line inside the loop is to retrieve the name and score for the current high score. Type the following:

```
//Get the name for the current high score position
name = highscore_name(???);
//Get the score for the current high score position
playerScore = highscore_value(???);
```

You need to replace the "???" with a value that represents which name and high score to retrieve. Remember there are 10 high scores with position 1 being the highest and position 10 being the lowest. Get your instructor to check the code before continuing.

Now add the following:

```
//Draw the name at the current x and y position
draw_text(xpos, ypos, name + ": ");
```

The score should be drawn at the same y position but its x position should be the current x position + a gap amount so that the score is spaced out from the name. Also the score is stored as a number but must be converted into a series of characters (a string) before we can display it. You need to work out what goes where "???" is. Get the instructor to check your code.

```
//Draw the score at current x position + a gap and current
//y position.  The score variable needs to be
//converted to a string.
draw_text(???,ypos,string(playerScore));
```

The y position must be shifted down by the height of the string before we loop back up and repeat the code to draw the next score. Also we must add 1 to the current high score position as well. Try and figure out how to do this and get the instructor to check the code.

Once you have written the correct code you can close the high score table object. Open the high score table room and add a single objHighScoreTable object to the room and then close the room.

Open the "objGhost" object and select the collision event with "objPacMan". Delete the "Exit Game" action and replace it with the "Go To Room" action from the Rooms group which will move from the current room to a specific room. Which room do you want to change to when you die?

We could have used the "Next Room" action to move from the current room to the next room. Can you think why Next Room is not the appropriate action?

Play around with position of the names and scores, changing the background, different colour for the text, etc.

Extras

The following exercises are for you to do yourself. Only outline instructions are provided. It is up to you to find a way of adding each feature to the program. Some may involve finding out about other parts of Game Maker we have not yet discussed. If you have ideas for different features to try instead, talk to us about it. To complete this session you must complete at least 4 of the 6 exercises, including one not marked as straightforward.

Fruit (moderately difficult)

The Pacman game also has an extra prize for the player. For a brief time during play some piece of fruit appears in the maze. (Cherries in the screen shot at the start of session 1.) If the Pacman gobbles that the player gets extra points.

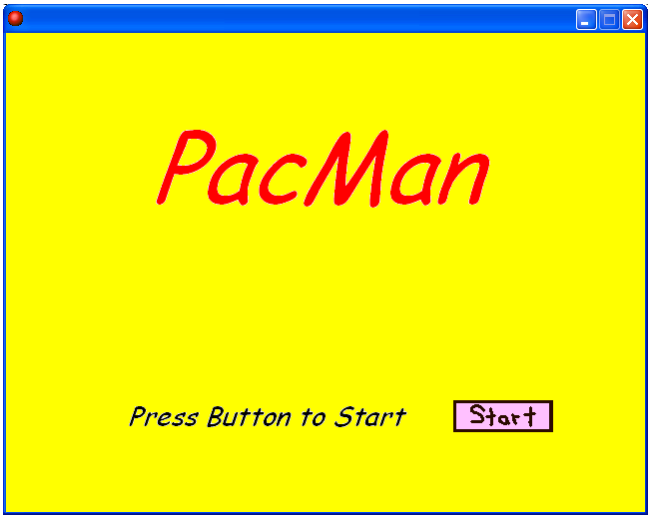
- Work out a way of implementing the fruit behaviour. You must decide what object(s) are involved; how to get the fruit to appear and disappear at a chosen time; and how to allow Pacman to eat the fruit.

Game completion (straightforward)

- Add instructions to notice when the level is over and allow the user to try again or quit. (The level is over when all the dots and all the energisers have been eaten).

Game starting (straightforward)

It is nice to have an introductory screen for a game. In Game Make this can be done quite simply.



- Create a room. Use Windows Paint program or some other graphics program to prepare a screen image welcoming players to the game. You could include some instruction text on the image Use the image as a background for your new room.
- Add a sprite that looks like a button (with "Start" on it)
- Create a "Start" object, and place it in the room.

- Add an action to occur when start is clicked which goes to the game room.

Game completion (straightforward)

A similar technique to the starting screen can be used for ending. When end of game is detected, after a suitable pause, jump to an ending room. It can be one with "Congratulations", or "Bad Luck, Try Again" as appropriate.

Extend High score table (straightforward)

Add buttons to your high score table to quit or play the game again.

Better ghost movement (challenging)

Can you find ways of making the ghost movement more interesting. Could the ghost move in the general direction of the Pacman (or away from the PacHunter)?

Lives (moderately difficult)

In most games play doesn't just stop after the player collides with an opponent. The player is usually given several chances – or lives. Find out about the *Lives* facility of Game Maker. Get your program to display the number of lives – in text, or preferably in a graphic form.

Help (straightforward)

Find out how the help facility works in Game Maker. Add a simple help page to the game.

Session 2: REVIEW PAGE **Name: _____**

Questions: Complete the following questions and be prepared to demonstrate techniques to your demonstrator. (You can then have your work verified.)

1. Is it a good idea to copy and paste displaying the name and score 10 times instead of using a loop? Explain your answer.

No, it's not a good idea to copy and paste the code for displaying the name and score 10 times instead of using a loop because it allows you to write the code for displaying the name and score once and repeat it 10 times, reducing redundancy. This approach enhances code readability, maintainability, and efficiency.

2. In a while loop why must you add or subtract from a counter variable at the bottom of the code inside the loop?

Because by updating the counter variable, you ensure that the loop progresses towards its exit condition. Without this update, the loop might become an infinite loop, causing the program to run indefinitely

3. In the code when displaying all the names and scores, why do you need to shift the y position down inside the loop?

Shifting the y position down inside the loop when displaying names and scores in the high score table is necessary to maintain vertical alignment and prevent overlapping.

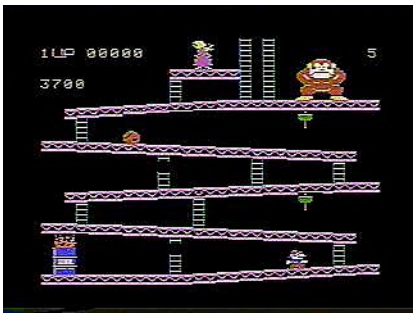
Verification: Your demonstrator will ask to see your game at this stage. You must show the high score table and one other extra.

Games – Pacman to Boulderdash – Session Three
Platform Games: Gravity and Tiles

A little bit of history from
http://www.informationheadquarters.com/Video_games/Platform_games.shtml

Platform games were a very popular genre of video games from the early 1980s to the mid 1990s, now all but forgotten by the majority of gamers and developers. Games like Tomb Raider may be seen as modern versions of platform games, and many of the old platformer franchises live on in 3D form.

Traditionally, the platform game usually scrolls right to left, with the playable character viewed from a side angle. The character climbs up and down ladders or jumps from platform to platform, fighting enemies, and often has the ability to gain powers or weapons. Later on, the term came to describe games where jumping on platforms was the main gameplay focus, as opposed to shooting being the main gameplay focus. These include games like Super Mario Brothers and Donkey Kong Country.

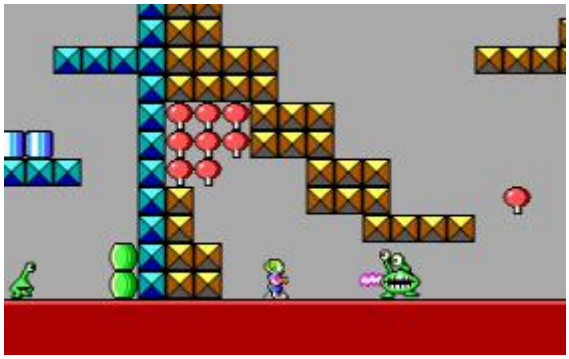


Chronology of significant platform games

* *Space Panic (Universal, 1980)*
* *Donkey Kong (Nintendo, 1981) – first game featuring Mario*
* *Super Mario Bros. (Nintendo, 1985)*
* *Mega Man (RockMan) (Nintendo, 1987)*
* *Haunted Castle (Konami, 1988) – predecessor of the Castlevania series*
* *Super Mario 3 (Nintendo, 1990) – highest grossing console game before Pokemon*
* *Commander Keen (id Software, 1990) – first major PC platformer*
* *Sonic the Hedgehog (Sega, 1991)*
* *Mario 64 (Nintendo, 1996)*

In Donkey Kong (upper picture), the little character (Mario, near the bottom right) runs along the platforms and climbs up and down the ladder. He can also fall off the ends of platforms, in which case gravity pulls him down, either onto a platform below, or off the screen to oblivion. Mario's goal is to reach the top, avoiding barrels rolling down. In Commander Keen (below) the picture is just part of a large play surface that scrolls. Players explore and solve puzzles.

Mark Overmars has this to say:



Platform games are very common, in particular on devices like the Game Boy. In a platform game you look at the scene from the side. The player normally controls a character that walks around in the world. This world consists of platforms. The player can walk on these platforms, jump or drop from one platform to the other, use ladders or ropes to get to different places, etc. On the platforms there are objects to collect, enemies to avoid or kill (often either by shooting them or by jumping on top of

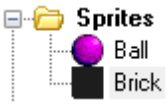
them), switches to press to open passages, etc. Also the player normally requires skill to jump over dangerous areas. In some platform games you see the whole level at once, but in most you see only a part around the character. In such a case, finding your way around becomes an additional challenge.

In this session we look at walking on a platform, using gravity, and some of the methods that are used to make the images more interesting.

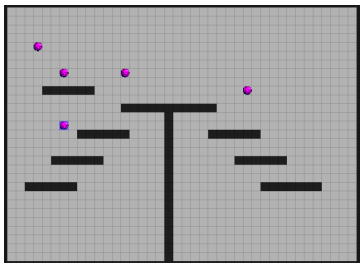
Platforms and movement

In the first section of this tutorial we will build some simple platforms and experiment with walking and jumping. We require some sprites. Whilst there is nothing special about platforms – any solid object will do – we need lots of platform surface in a practical game. For these experiments we will build platforms out of small 'bricks' – just little squares. We will also need a sprite for a character to move about. In this experiment we will use a ball as a 'character', because it nicely demonstrates positioning problems that can arise.

- Start a new game called Platform
- Create sprites for Brick and Ball, using the files from 'L: in COMPX251\Platform. Ball is straightforward. Make sure the Collision Mask setting is 'Precise' for the ball sprite.
- Having loaded both sprites you can see that the brick loaded correctly by looking at the left panel. Both sprites can have precise collision checking, although for the brick it doesn't matter (why?).



- The Ball we have supplied is 32 by 32 pixels which is too large. Use the Resize Sprite button to scale the sprite to 16 x 16.
- Create objects objBrick and objBall using the sprites.
- Make both objects *solid.*
- Open the provided toom and set the size to 640 by 480. Make sure the grid setting is 16 by 16. Change the background to a grey colour (you may want to change the colour of the grid lines in the Grid Options).
- Use the Brick object to make some horizontal lines (platforms) at different levels. Make sure that one goes across the bottom of the screen, so that your character cannot fall out of the game. Also make some vertical lines, 'holding up' one or more of your platforms. These will function as barriers, so that the character must jump around the platforms to get from side to side of the screen.
- Put a few Ball objects into the scene. Try putting them at different heights, so that when we set 'gravity' to act on them, they will fall to the nearest platform.



The picture on the right shows my test setup.

The next step is to add movement. Yet again we will demonstrating a different way of moving characters. You shouldn't be too surprised. Movement and interaction are the most crucial aspects of game play. In the Pacman game, characters were moved by setting a direction of movement and letting them continue until they hit a wall or were told to change

direction. In this game we will use two different movement techniques. For jumping and falling, we will use a 'move until you hit something' method. For left and right movement we will have the character jump every time a keyboard event happens, resulting in a combination of jumping and falling behaviour.

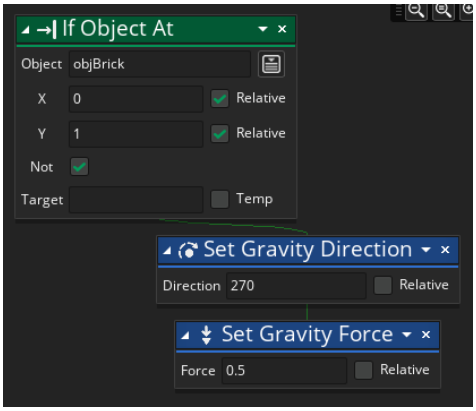
Falling

Falling is easy. Game Maker allows us to let 'gravity' act on objects. Unlike the real world, we even have a choice as to the direction that gravity acts in. The purpose of falling, in a platform game, is to get a character back down to a platform when it has jumped or stepped off the edge of a higher platform. Gravity is provided as one of the motion options – what it really does is accelerate an object. We apply gravity to a character if we want it to fall. It will turn out that we have to turn gravity off when the character is on a platform, so we can't just leave gravity on all the time. So the first thing to do is to notice when a character is in the air, and turn gravity on for it. The logical place to do this is in a step event – on every step of the game our character will look to see if it is on a platform. If not, it will turn on gravity, and start falling.

- Add a *Step* event to the Ball object.

Now we need to check if there is not a brick object directly beneath the ball. If that is true then set gravity to on. In other words we test immediately below the current ball position. Gravity setting requires a direction as an angle. 0º is to the right, 90º is straight up, 180º is to the left, and 270º is straight down. The gravity value represents the strength of the force. The value 0.5 works quite well, but you should experiment with other settings of both direction and gravity.

Add the If Object At action () from the Collisions group in the toolbox. Set the Object to check to 'objBrick', tick the 'Relative' option for both X and Y and then set the Y value to 1. This means it will check 1 pixel below the ball relative to it's current position. Can you figure out how to make it check if there is NOT a brick object below the ball? Then attach the 'Set Gravity Direction' action from the Movement group to the IF action and set the direction to 270. Then attach the 'Set Gravity Force' action from the Movement group under setting the direction and set it to 0.5.



- Try the program, the balls should now fall nicely. Try different direction and gravity force values.

The next task is to stop it.

- Add a collision (with objBrick) event to the objBall object.
- Stop the ball falling by setting it's speed to 0.

What happens and why (question 3 on the review page)?

The landing is a little odd, but before fixing it we will add horizontal movement and jumping.

Horizontal movement

All the normal options for moving a character from side to side with the keyboard are usable here. You can experiment. Some difficulties of interaction with the vertical movement will occur with some techniques. Partly for the sake of trying something different, and partly to avoid trouble with vertical movement, we will use 'move by jumping'. 'Move by jumping' will work as follows. When a <left> Key Down event occurs the character jumps left (minus x direction) by 4 pixels (or more for a higher speed). When a <right> Key Down event happens it will jump right (positive x direction). Because Key Down events are sent at every step while a key is held down, the character will move while a key is down and stop as soon as it is released. Rather than use collisions to notice if the character tries to walk through a barrier – we will check before each jump to see that the way is clear. That will leave the platform collision just the task of stopping falls.

- Add a <right> Key Down event to Ball
- Add an action: *Jump To Point.* The position is x = 4, y = 0, relative for both x and y.
- Try the program. Press the right arrow key.

Nothing should happen, unless a ball is in the air. Why doesn't the jump work when the ball is on a platform? The reason seems to be that gravity is still on! The ball is trying to fall through the platform and continually being stopped by the collision event. Collisions stop all motion in a step, so the jump gets prevented as well as the fall.

- Add another action to the ball's collision with brick (in addition to the stop): Set gravity off (same as set on, but make the *gravity* value zero).

You should now find that the program responds to the right arrow key. Interestingly, the collision already inserted serves to stop the ball from passing through a barrier of bricks from the side – so the idea of looking ahead to see if movement was possible isn't needed.

- Add the corresponding event and action for leftward movement (move -4 pixels). You should now be able to freely move the ball left and right.

Jumping

Pressing the up arrow key should make the character jump. Jumping can take advantage of gravity. All we need to do is start the character moving upward. The speed will dictate the height of the jump.

- Add a **Key Pressed** event for the up arrow to the Ball object. (Review question 5)

- Add the 'Set Direction Fixed' action and the direction should be up. Add a 'Set Speed' action afterwards and try different speeds. A value of about 10 works quite well on my platforms.

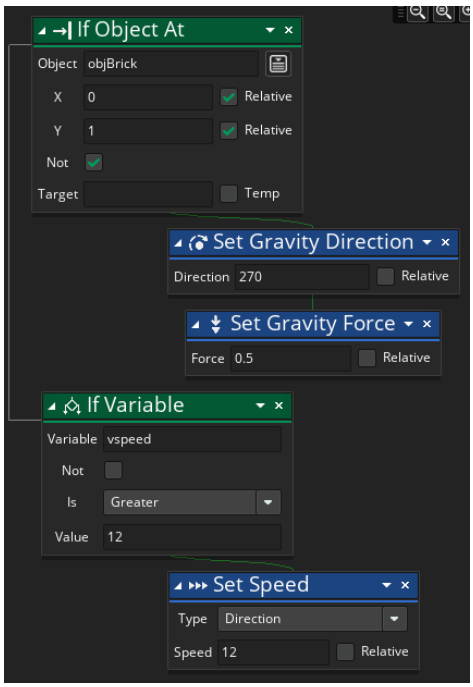
You should now be able to navigate, jumping and falling from platform to platform. There are still a number of options with the movement. Notice that pressing <up> while the ball is in the air causes an even higher jump. Also, the left and right keys can be used while the ball is in the air. This is all unnatural. The most common resolution in platform games is to disallow extra jumps by having the <up> key action only perform the jump if the object is on a platform. Usually the left and right keys continue to function in the air.

- Change the <up> Key Pressed event which checks if there is a brick object under the ball and if there is then move up and also at the speed you have chosen.
  Hint: Use 'If Object At' action just as when we turned gravity on.

Aside: We have not programmed an action for the <down> button. Many platform games used the <down> button to make the character duck – an action that could be used to dodge missiles flying at head or chest level.

This just leaves us to limit the speed of falling. You will probably not have experienced any problem, but if a character falls too fast it can fall right through a platform (why?). Although it spoils the dynamics a little, it is normal to limit speed of falling.

- Add an action to the *Step* event of Ball, that checks to see if its speed is greater than some value, and if so sets it back to that value. A limit of 12 works quite well, but you should test with a smaller value to make sure it is working. You will have to use the 'If Variable' action () and type 'vspeed' into the variable textbox and check if it is greater than 12. Then set the speed to 12 if that expression is true. (See the diagram below).



Precise collision detection

The only remaining issue with movement is that the ball can sometimes end up in rather unlikely positions. This is a consequence of the precise collision checking used with the ball. If you turn off precise collision detection you will find that it doesn't behave this way. With a ball the results are not visually great. However, with a character that is reasonably nearly rectangular – and has nice wide feet – the visual effect is acceptable. This seems to be the best solution – characters don't have exact collision detection. It also avoids problems that can arise from changing sprites as we will do shortly, so long as the sprites are the same size.

- **Change** your sprites to make your character sprites not use a *precise collision mask, set it from Precise to Rectangle.*
  Try that now with the ball.

Making it look better

Plain black platforms are not very interesting. It would be possible to build up lots of detailed sprites to make a better scene, and sometimes this is done. However, it leads to a huge set of objects and is hard to manage. The easier solution is to find some way of painting a scene and then to position platform bricks invisibly over it. Two methods are available in Game Maker. The first is to use a background picture, even a photograph will do. To demonstrate the principle we will use the people picture from the home page of the computer science department web site (some years ago). The game will be to try to jump from head to head.

- Make a copy of your game by selecting File>Save As and call it HeadToHead
- Working in the HeadToHead game, delete the room and create a new one with a width and height of 640 x 480 and make sure the grid size is 16 x 16.
- Make a background sprite using the picture in people.jpg (provided) and place that sprite as the background to the room. You may want to change the colour of the grid lines to make them more visible.

You can stay with the black brick for this exercise, but I found it difficult to see against the picture. For that reason I changed the brick sprite to yellow.

- Put platforms over the heads.
- Add a Ball over one of the heads.
- Try to play the game

Often in platform games, just doing the jumps is part of the challenge, so it is ok to have some hard jumps. They should be possible though.

Yellow bars in people's heads don't look very good. The trick is to make the bars invisible.

- Turn off the *Visible* check box in the objBrick object and try again.

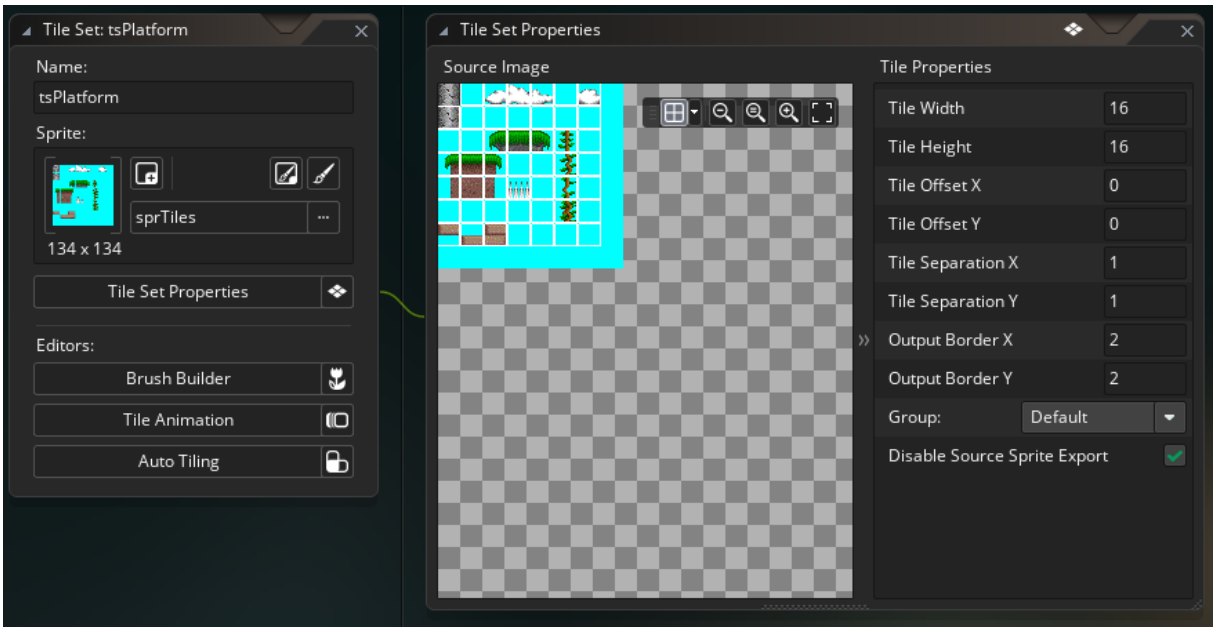
Hmm. This isn't going to be a smash hit. It wouldn't be so bad if people really had flat heads, but it does play, and we do have a scene rather than black bricks. I hope that this demonstrates the principle. Clearly a carefully drawn scene with flat places for the platforms would be nice. It is also possible to have platforms that are not flat (see later), and fit the heads a little better.

- Save your game and and now select File>Save As to create a new copy of the game and call it "PlatformTiles" to experiment with the next way of providing background.

Tiles

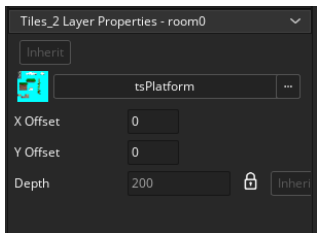
An alternative to a background picture is to use tiles. The idea behind tiles is that of having a collection of little graphic images that can be assembled in different ways to make scenes. Tiles are rather like sprites, but can be used by themselves, without involving objects. For example, we could build a garden scene by placing a selection of tree images on a suitable background. In GameMaker we take a picture and break it into little squares (tiles) and place the tiles in our room. Tiles are simpler than objects. They are just graphic items. They cannot move or have collisions. From a game performance point of view they are inexpensive. We just put them on the screen and nothing further needs to be done with them. The platform demonstration from the Game Maker web site has an example tile image. It is included in the Platform directory as tiles.gif.

- Create a new sprite and load the tiles.gif file and name it 'sprTiles'.
- Then create a New Tile Set called 'tsPlatform' and set the sprite to 'sprTiles'.
- To improve the appearance, set horizontal and a vertical separator values to 1.



- Remove your existing room and create a new one and make sure the grid size is 16 x 16.
- Choose a pale blue colour as the room background
- Click on the Create New Tile Layer button ().

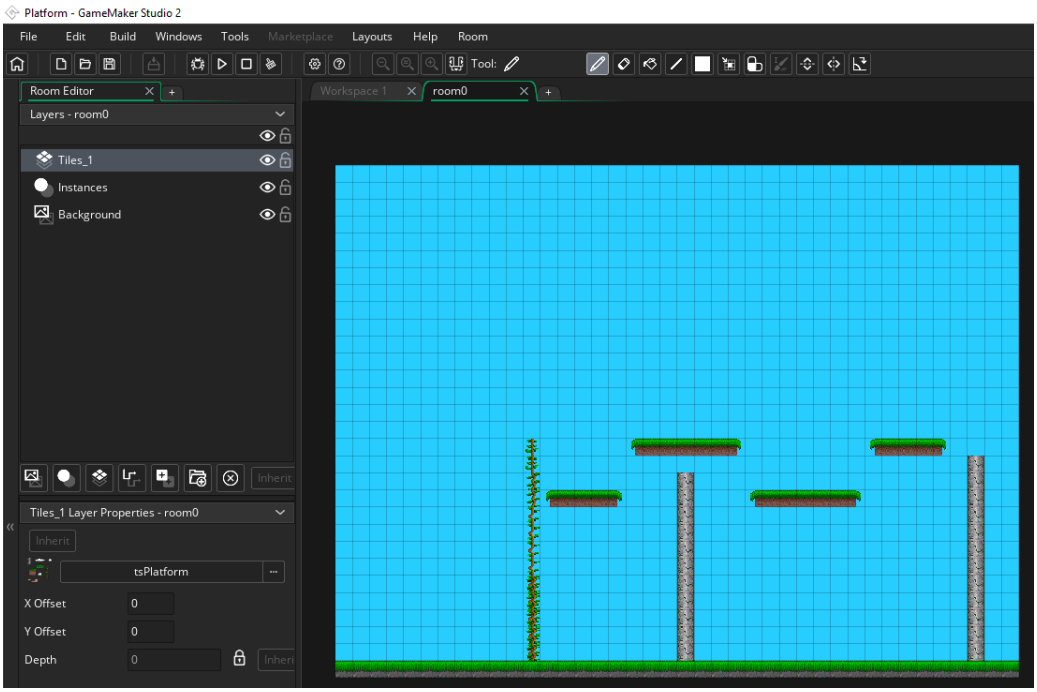
- Select *tsPlatform* as the provider of tiles. None of the default settings should need changing. You should see a preview of your tiles on the right hand side. Use the zoom in and centre fit buttons to make the preview bigger.



Notice in the preview the very first tile is blank instead of having a pipe image. Game Maker studio always reserves the very first tile for a black tile so when creating or using tilesets you need to be aware of that. That is why there are 2 pipe tiles in the original gif file.

You can now draw with tiles – select a square from the (small) tile image, and place it in the room by left clicking (left click and drag will add multiple tiles), right clicking will remove a tile from the room. If the background of the tiles doesn't match the background of the room it will look odd. Go back to the sprTiles sprite and edit it and remove the background colour. Now each tile only has the tile image and the background is now transparent.

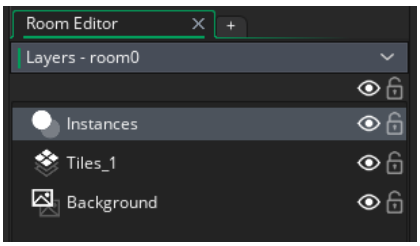
The tile image breaks up nicely to allow you to assemble grass covered platforms and ropes of creeper for climbing. The plain brown blocks make good barriers. Note that you cannot put tiles over the top of objects. This whole system will work in the same way as a single background picture. The tiles provide a *kitset* for making up scenes.



When you have the scene, select the Instances layer to place instances of brick objects over the platforms and barriers. If they are set to be invisible, the user of the game sees just the

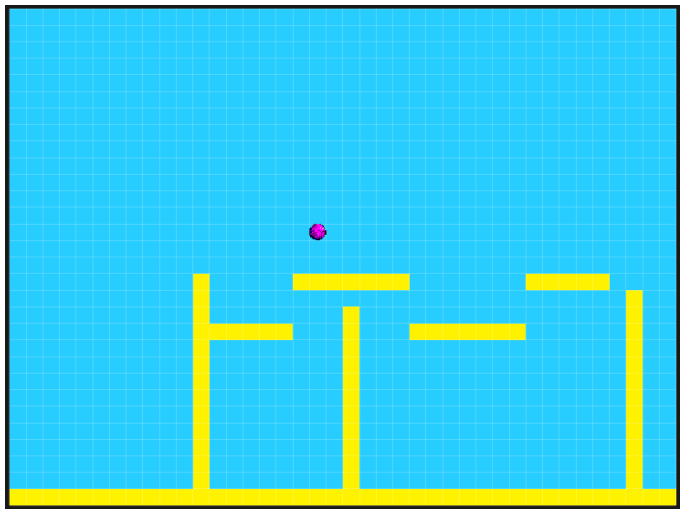
picture made from tile segments. The game plays on the invisible platforms. Notice that the objects are behind the tile:

This is because the Tile layer is before the Instances layer in the Layers panel. Drag the Instances layer before the Tiles layer:



Now you can see your objects placed on top of the tiles as you add them to the room. Feel free to put the layers in the order that you prefer.

My scene with bricks over most of the platform tiles:



- Make your own scene using the tile system. Make sure to put in a good high vine so that there will be something to climb (when we add climbing).
- Experiment. Satisfy yourself that it all works (Don't forget to add a ball).

Improving the character

The Ball is, of course, not a very good character. Usually the character is a person or animal. Platform games vary in the amount of detail programmed here. The simplest solution is to use a single image of the character as we have with the ball. The next idea is to have different sprites for each of the movements the character carries out – as we did with Pacman. We require left and right facing views. If we want to make the character look as though it is walking we can use animated sprites – one for walking left and one for walking right. When using animated sprites it is a good idea to have a third form to represent the idle player – otherwise it will appear to be walking on the spot when standing still. Other sprites can be used as well. A version that waves its arms and legs is good for falling. Later we will add the

idea of climbing, and separate sprites will be needed for the character climbing up and down, or just remaining still on the ladder.

For initial experiments we have provided three very crudely animated sprites: PersonLeft, PersonRight and PersonIdle.

- Create sprPersonLeft, sprPersonRight and sprPersonIdle sprites . You will need to edit each sprite and then multiple select all the files and then remove the background. It will remove the background from each sprite. Make sure you only have the black outline of the person. Set the speed of the animation to 20 and then click the Toggle loop/pingpong button to set it to loop mode and press the play animation button underneath to preview the animation of each sprite.
- Create a single person object – best to duplicate the objBall object and the sprite should be set to sprPersonIdle.
- Add actions to change sprites to the appropriate form (remember Pacman directions).

Run and test the game, what happens to the sprite if you hold down the left and right arrow keys? Notice that it does not animate when you hold the keys down. Go back to the <Left> event and open the Change Sprite action. Notice that the frame field has the value 0. The frame value tells the game which frame of the animation to play.

When you hold the left arrow key down then this event is triggered all the time and the frame is always set back to 0 which is why it doesn't animate properly. What we need to do is tell the game to set the frame value to the current value so that it moves through all the frames of the animation.

For the frame field type in **image_index**. This is a variable which stores the current frame of the animation and the game automatically updates this variable as it cycles through all the frames in the sprite.

Run and test the game and make sure that the left animation is now working correctly and then fix up the <Right> event. But we still have a problem, what happens when you release an arrow key? Notice that the animation does not go back to being idle. Work out how to fix this.

The sprites provided are not very good – they are inclined to get stuck in platforms as their geometry changes with the sprite swaps. It is necessary to be quite careful in making animated and alternative sprites – that their borders are in the same places.

- Experiment with sprites. You can create better ones than I did, based on PlayerLeft and PlayerRight files, also supplied in the Platform directory. This image is wearing a long dress, which covers the feet, and so animation of walking is not necessary.

Tiles and background images provide two ways of doing the same thing. In fact, you can mix and match as you choose. It is quite sensible to draw a background, then use tiles for the images in the foreground. Just as backgrounds can be behind (normal) or in front (foreground) of the moving pieces, it is possible to place tiles both in the foreground and background.

Scrolling and the play view

The following is quoted from Mark Overmars.

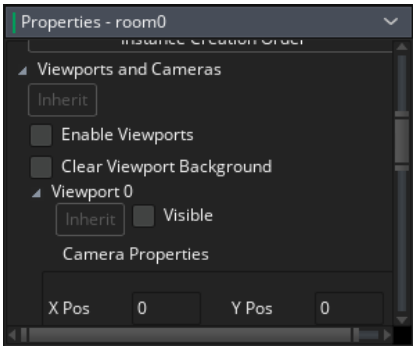
"Up to now we always showed the entire room. For many platform games this is not what you want. Instead you want to see only a part of the room, around the character you are controlling. This makes the game more challenging because the player must try to detect his way through the platform. You can also hide prizes at difficult to reach places in the room.

Fortunately this is extremely simple to achieve in Game Maker. When designing the room, click on the Views tab. Click on the checkbox Enable the use of Views to start using views. Select the first view and check the box Visible when room starts to make sure this view can be seen. Give it a width of 300 and a height of 200 (or something else that you like). (As we are going to let the view follow the character there is no need to specify the left and top position of the view in the room. Also, because we use just one view, we don't have to specify the x and y position of the view on the screen.) As the object to follow we choose (at the bottom) the character. The view will now automatically move to keep the character in focus. We don't want the character to get too close to the border. To this end we set the Hbor and Vbor values to 64. There will now always be a 64 pixel area visible around the character. Finally, to get a smooth view motion we set the maximal view speed to 4. (This also gives a very nice effect at the start because the character comes slowly into view.)"

See the video link posted on Moodle in the Games Theme section for a video which explains how Camera and Viewports work in the game before proceeding.

Having the view is nice but in this case it makes the window rather small. In a real game program it would not be a problem because we would want a big game for the user to explore. Nevertheless, let's try it out.

Go into your rom and scroll down the room properties until you see the 'Viewports and Cameras' section. Click the arrow to unhide that section. You should see this:



Tick the "Enable viewports" option. Then click the arrow next to Viewport 0 to see it's properties. Tick the 'Visible' option and set the camera width to 320 and the height to 240. In the Viewport properties, set the width to 640 and the height to 480. You will notice a bounding rectangle appear for the view in the room preview.

Run and test the game, what can you see?

You should only be able to see the top left area of the room. Go back to the Viewport properties in the room. We want the view to follow the player as it moves otherwise the player could walk outside the view and we wouldn't be able to see it. Figure out how to make the view follow the player. Move the player to an area of the room so that it can move freely to test out the view. Run and test the game and you should now see this work. But there is also a problem as you have to be very close to the edge of the view for it to start following the player.

Go back to the Viewport properties of the room and the values we want to change are the Horizontal Border and Vertical Border values. This sets how many pixels are always shown around the player. The higher the number the more space around the player. Set both values to 100 and see what effect it has on the game. You can experiment with these values when creating your game to see what works best to fit your vision for the game. Test the game, also place your person on a platform to see how the view moves when the player jumps.

Before carrying on go back to the Room properties and untick 'Enable Viewports' to make it easier to test the other parts of the game.

Climbing

Platform games nearly always have ladders or vines or ropes that the players can climb up and down. We can make the character climb by extending the motion events and actions. First, a ladder is a little different from a platform. It should be possible to walk past the bottom of a ladder without colliding and being stopped. This can be arranged by having a different kind of brick (it is best to make it a different colour to the platform bricks for ease of editing). The new brick will be used in the same way as platform bricks, but to identify the location of ladders.

You can add an If Object At action to the end of the Step event code in the player object which checks if the player is on the vine or ladder. If that is true then set gravity to 0 and set the speed of the object to 0. Now you can jump onto the vine or ladder and stay there.

Then you can add an <Up> 'Key Down' event to the player object and check if the player is on the vine or ladder and if that is true then set gravity to 0 and set the speed to 0 just to be sure and then use the 'Jump To Point' action to move 4 pixels up. Use the same principle to do movement down the ladder.

Finally, climbing looks best if a suitable *climbing* sprite is provided.

- Extend your game to allow the character to climb the vines from the tile set.

Sloped Platforms

The very first platform game (arguably), Donkey Kong (see screen shot in the introduction to this session) had sloped platforms, and barrels rolled down them. The subject of objects that move about is part of the next session (where boulders fall). However, sloped platforms and platforms with other strange behaviour are a feature of platform games. To implement a sloped platform the left and right moves must be made more sophisticated. One way of doing it is to check to see whether the play piece is on a sloping platform (yet another kind of brick, but it can have a normal platform brick as a parent, to avoid too much rework), and if so

implement left and right movement as a jump left or right and also slightly into the air. The falling under gravity feature will take care of the rest – giving a bouncing motion along the slope. To have the character slip downwards on a slope – the *Step* event must check for the character being on a slope and do a small jump sideways in the downslope direction. Again a nice bouncy motion will result.

A strange platform behaviour that is quite fun is the trampoline. When a falling piece collides with a trampoline it can be immediately set to move up again. It is even possible to provide *elevators* by making gravity work in reverse in some areas of the screen.

Finishing off

This session has concentrated on the mechanics of making a platform game. We have said very little about game play. To build an interesting game around the platform idea we can add all or any of the following.

Monsters: A monster can move. Usually they are set up to walk backwards and forwards on platforms or to fly backwards and forwards in the air. Monsters may also shoot at the player, and the player may be given the capability of shooting back.

Crushers: An object (like a rock) that bounces up and down makes a nice obstacle. The user must time their movement to avoid being crushed.

Pits: Usually there are dangerous parts of platforms that the user must jump over. To implement this feature, make yet another kind of invisible brick – one that sits at the bottom of pits. The player dies when the player collides with it!!!

Prizes: Things to collect – much like the dots and fruit in Pacman.

An Exit: Usually somewhere on the play surface there is an exit point that represents the end of the game (or the level). It may not be accessible until puzzles are completed or all prizes collected.

Puzzles: Usually these take the form of implements left lying around. For example there may be a switch somewhere. The user must find it and as a result: some other action or access to some other place will be enabled. The easiest way of allowing the player to operate a switch is just to notice a collision. Colliding with a switch could open a door (by removing a door object); create a new object, whatever. The user might have to push blocks around to provide access – but that's the subject of our next session (Boulderdash)

- Add one more feature – try a sloped or trick surface, or one of the suggestions above.

Session 3: REVIEW PAGE **Name:** MIN SOE HTUT

Questions: Complete the following questions and be prepared to demonstrate techniques to your demonstrator. (You can then have your work verified.)

1. Why don't we make bricks transparent in the first experiments?

   Because we need to see where we are jumping and without the background we will not know where is the brick

2. Why do we make brick and ball solid?

   So they will not collide each other

3. What happens to the ball when the platform collision action is just 'stop moving'

   the ball cannot jump anymore

4. What happens if two balls collide when doing left or right movement? Why?

   it will show as one ball because we didn't put collision between balls , if we did the ball will be bouncing on top of each other.

5. Why is upward motion programmed with a key down event rather than a key press action?

Because key down event make the player move one jump at a time. if you use key press the character will get stuck on the tree

6. How could you prevent left right movement while a character was in the air? What are the consequences? What further changes would need to be made?

While there is nothing below the character cannot move.
If you create it correctly there will not be any problem
we need to put if event in step and crate a few events handling.

7. How does the character act on collisions - from above, below, sideways, whilst moving?

If there is a brick set gravity and force and speed to 0
Tree you can move up and down or jump off the tree
while moving you cannot jump again

8. What feature did you add?

Change the direction left and right to no animation
More tree and more platform

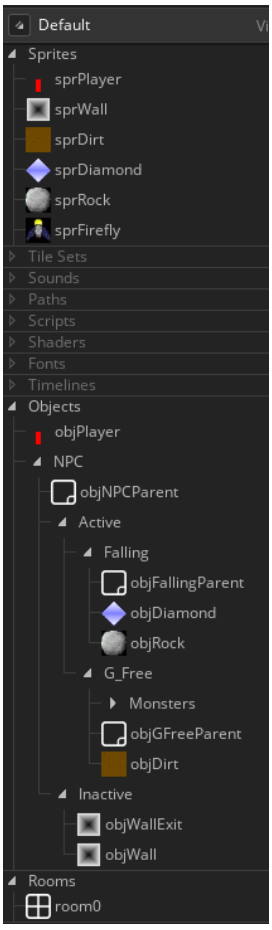
Verification: To assess your competency of this material your demonstrator will verify that you have completed the above questions, and can do the following:

1) Change the maximum speed of the ball object to 5.
2) Create a new room and use the tiles to add some platforms to the room.
3) Show that your person sprite goes back to being idle when an arrow key is released.

Games – Pacman to Boulderdash – Session Four
Boulderdash: A Classic Favourite

Now that you have lots of experience of making games using events and actions, it's time to make one of the classic games of it's time. This quote obtained from a website specializing in older games (http://www.myabandonware.com/game/boulder-dash-3h) gives some history.

"In 1984, First Star released a low profile, unassuming puzzler called Boulderdash that would become the company's best selling series and sparked numerous clones even decades after its release. The game could perhaps best be described as Pacman with brains. The concept is simple: help guide adventurous Rockford through 16 caves packed with tricky puzzles to solve. Your basic goal on each level is to collect the specified number of diamonds to advance to the next, more difficult, level. Along the way, you will foil fireflies and butterflies, trap bubbling amoeba, and transform worthless boulders into valuable diamonds and vice versa. Of course, it is not as easy as it first seems, especially since Rockford can be instantaneously crushed by boulders if he carelessly tunnels right below them, opening up space for them to fall down. The trick is to carefully tunnel your way to all the diamonds without making the rocks fall on you, or quickly evade them once they do. The game is a lot of fun for its time, and hundreds of levels kept many fans glued to the screen for hours on end."



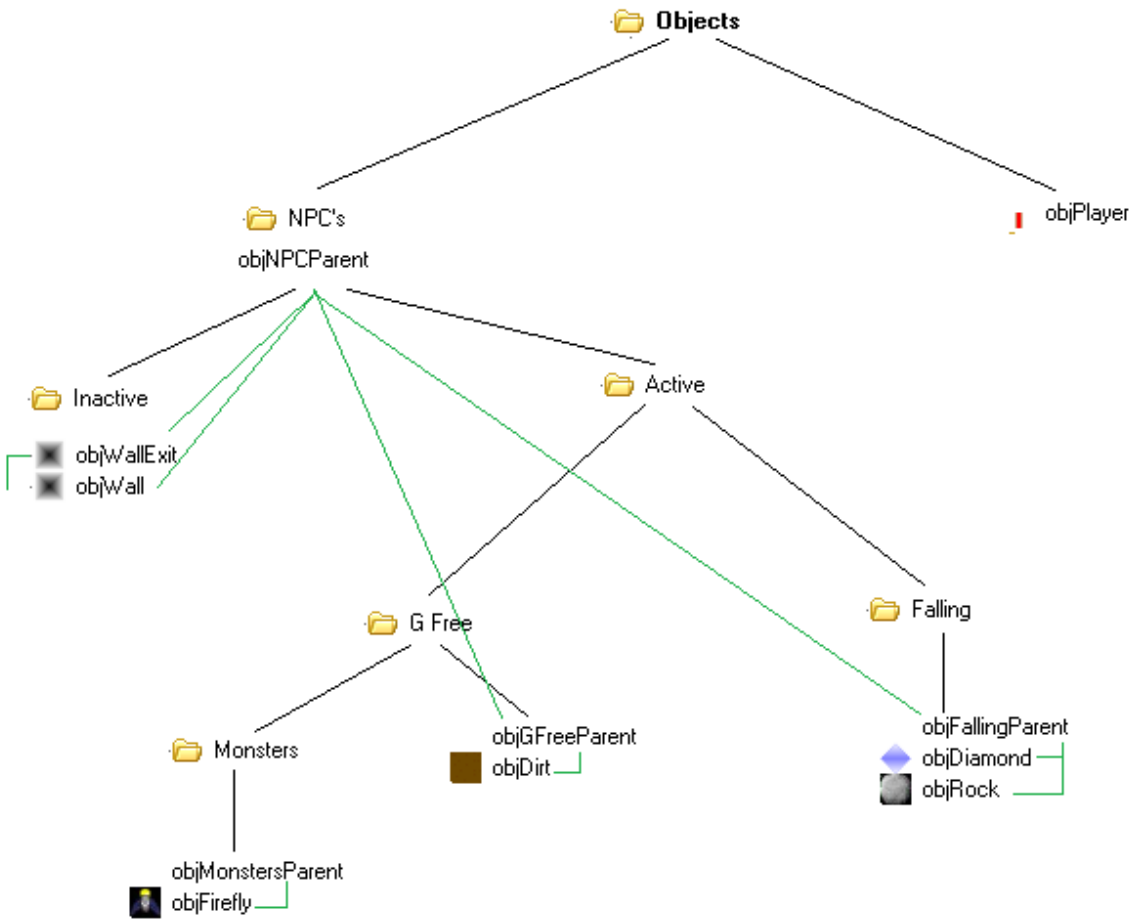
The objective of this part of the module is to make a clone of this addictive game. You will find in L: in COMPX251 a folder called "Boulderdash". Copy that folder into your account and open the game maker project that is inside it. You will see that all the required objects, sprites and backgrounds have been loaded for you. If you open the tree, you will also notice that the organization of the objects in the side window has changed. Click on the little arrow next to "Objects", and instead of seeing all the objects pop out at you, you'll only see one object, and another group. All of the objects have been arranged into groups in a way that reflects their parenting.

Going through the object tree, we have the player, currently represented by a 4 sprite animated gif file with a red line pointing in the direction the player is facing. The group NPC (which stands for non player character) contains all of the objects that the player has no direct control over. In that group, there is the NPC Parent, of which every object is eventually a child. There are Active objects, and Inactive objects. Active objects are the things in the room that the player interacts with frequently. The Inactive, obviously, are the objects that the player tends not to interact with. Each of the particular objects, including the objects that have no sprites will be looked at in further depth when you come to use them.
So what does this game need to do? For those of you who have never played the game you may wish to invest some time in playing one of the many free clones available on the internet, but I'll describe it here as well.

The game of Boulderdash requires the player, who can move in one of the 4 compass directions, to collect diamonds in the mine that he is currently in, avoid being squashed or

trapped by rocks, being eaten (lose a life) by the monsters in the mine, and getting to the exit in the time allotted. Some levels require speed, some are puzzles, and some are experimental levels where you get to play with a new object that has been introduced.

Here is a tree picture showing how all the objects are parented. NOTE: Please do not change the parenting structure unless told to, or you have a clear complete understanding of how the object parenting hierarchy works. The parenting is important in minimizing the number of events and actions that need to be entered.



If you follow the light coloured lines upwards, it will show you how each object is parented to the next. This information will be useful in entering events, as you will see later.

Ok, on to making your character's events, and making it move.

Step 1. Making the character move

Like Pacman, Boulderdash uses a grid. You may have noticed that all the objects are 32x32 pixels in dimension, and that the play room is set up in 32 by 32 squares. Unlike Pacman, the player doesn't move smoothly. In the classic Boulderdash game, the character moved a square at a time instantly - it did not progress smoothly from one square to another. Therefore using Game Maker's speed and direction functions for character movement would be quite cumbersome. It would be necessary to set and reset the speed on every step, and make sure that the character snapped to the grid accurately. Instead, we will do movement in

Boulderdash by simply making the character jump 32 units in the appropriate direction whenever a movement key is pressed, and the way is clear for movement.

The interest (and complexity) in the game results from interacting with each of the objects that Rockford meets in the appropriate way. There are quite a number of details to take care of, as usual – game development is all about taking care of details. We will start by programming the basic movement, and add in the detail bit by bit.

A bit more about animated sprites (GIF files)

As in Pacman and the Platform Games we want to have our character face the direction of movement. As well as improving appearance, this is good user interface design. Even when a movement cannot be completed (walking into a wall, for example) the change in the character shows the user that their command has been recognized. In both Pacman and the Platform game we used different sprites for each direction. There is no reason not to use the same technique here, except that this is an opportunity to experiment with another capability of Game Maker. Instead of using separate sprites for each direction of movement, we will use a single sprite. The sprite will be built like a four frame animation, but the four frames will be the four images we need. This method is just a little bit more convenient than using separate sprites, in some situations – we don't have to have separate sprites for each image.

In Game Maker we can either use the pictures as an animation, as we have done, or just as a little 'library' of pictures. It is this latter, library like, feature we will use in this game. You can designate which particular picture you wish to see by setting the index of the picture you want. I.e. You can ask for picture 2 (the third picture along, since most computer numbers start at 0, not 1), or picture 7.

In Game Maker, each object has information stored with it. Each item of information has a *variable name* and we can use the information by using its variable name in formulae that we enter into actions. We have already seen the 'direction' value associated with an object, and used it as a 'value' in a move action. There is another value associated with objects, named 'image_index' (yes that really is an underline character between image and single, and it is part of the name). This variable tells the game which particular picture to use to display the object onscreen. Any number 0 or above will set the picture to be static (non changing)). There is a kindred variable that is called 'image_speed', which sets how many frames per step are shown for the object. i.e. If image_speed is set to 0.25, it will animate at ¼ frame per step – or one step every 4 frames. This allows us to slow animations down, if image_speed is set to 0 then no animation is shown.

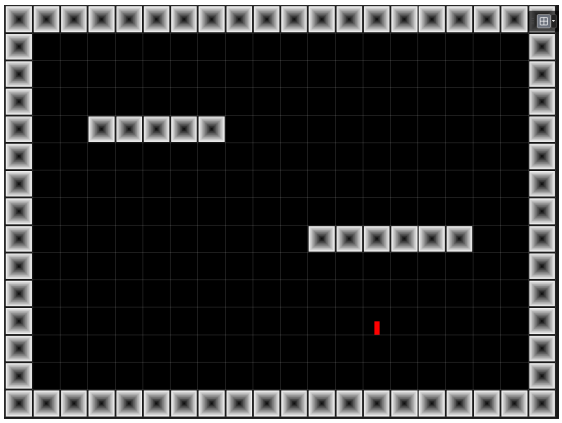
In Boulderdash therefore, there is just one sprite associated with the objPlayer object and it's image speed is already set to 0. It holds the four images, facing different directions. (Try looking at the sprPlayer sprite in the Sprite Editor.) It can be used by setting the particular sprite image desired whenever a direction button is pressed. So, when you press the down key, the object will choose a sprite that you set to represent the player facing down.

- Set your character to have a particular sprite when the game starts by setting the variable 'image_index' to 0 (in the character's *Create* event). To set a variable, add the 'Assign Variable' action () from the Common group in the toolbox.
- Make key press events for your character to move in all four directions.

- Put actions to jump 32 pixels left (x = -32, y = 0, relative), right, etc into the key press events
- Set the variable 'image_index' to 0, 1, 2, and 3 for each of the four directions (Down, left, up, and right respectively) when the character moves.

In the same way, the sprWall sprite actually has two images.

- Set the variable 'image_index' for the objWall object (in a *Create* event) and choose either 0 or 1, depending on which you prefer.
- In the room add, titanium Walls all the way around the outside, and a few Wall blocks inside the room. Place an instance of your player in the room. Make sure that movement works.



Now, the character is able to move, but has no restraints. He's able to pass through the titanium walls (which in the original game are indestructible), so there needs to be some checking done.

Since titanium walls are defined as solid objects,

- Add a test action make sure that the player isn't going to move his character into a wall object before moving it. [Use the 'If Object At' action and check to see if there is not a wall object to the left of the player.]
- Do that for all four directions.

So, now your character is able to move around a room bounded by the titanium walls.

- Test your player's movement by moving him round the room. Does he face in the right direction when you press one of the movement keys? Can he move through the titanium walls? Try from all directions.

Next, you need to add the dirt/mud to the level. Remember Rockford is a miner. We start a screen with soil/mud/dirt/matrix (whatever you want – we have set up the object objDirt). Usually each square in a room (that doesn't contain other objects in the game such as diamonds, rocks, monsters and so forth) is filled with dirt. Fill your level with Dirt objects.

When the character moves, it leaves a clear space behind – Rockford is making tunnels through the dirt. So, what actually happens is you move into the dirt, trigger a collision between the dirt and the player, and destroy the dirt.

- Fill your level with the Dirt object, but leave your character and the titanium walls. Don't add a dirt object where the player is.
- Implement the event where the player collides with the dirt and destroys it, leaving a trail of clear space behind. Test!

Optional Challenge

You will notice that even though in the original game, it was only possible to move in 1 of the 4 directions, if you are quick enough, you can make your character move in 8 directions (the four standard ways plus the diagonals).

- See if you can implement a method where your character can only move in one direction at a pace of one square per step.

You now have the most basic parts of Boulderdash. There is, as of yet, no objective, and nothing to interact with other than the mud and the walls. What the game needs is a reason to play it. The first way to introduce this is with the exit.

Step 2. Turning your program into a game.

You will see, if you have a look at the inactive objects, that there is another object that looks exactly like your titanium wall. This is your exit point, and is made of titanium as well, so it must not be destroyed (for obvious reasons). A way of indicating that the door is open is needed and you need to open the door when the player has completed the objective of the level – ie: picked up all the diamonds.

- Make a creation event in your 'objWallExit' and set the 'image_index' variable to 1 so that it looks different from a normal wall. If you wish you could create a separate sprite which is animated and loops through both images so it looks like the wall exit object is blinking.
- Now go to your play object and add a step event to the player object. Add the actions that will get the number of diamonds in the room and the test that value to see if it equals 0. If it does then we need to destroy one of the wall objects and replace it with the wall exit object (or you could just place the exit object somewhere in the room). To your If action add the 'Destroy At Position' action () and type in the x and y position of one of your wall bricks. Then add the 'Create Instance' action and create an 'onjWallExit' object at the position where you destroyed the wall object, **don't tick the relative options because we want to use a specific position**.

Run your game, and if everything is working properly, your exit should be appear. Why? Because you have no diamonds in your game yet. Place a single diamond anywhere within the room and run the game again.

Now that you have your character that can run around the screen, leaving a trail of space behind him, you need to do some more work on the objects that your character can interact with, namely the diamonds and the boulders.

Step 3. Boulders and diamonds and falling, Oh My!

The first thing that needs to be done with the objects that the character interacts with is to define their behaviour. You can add to the list or remove as you go, but you need a solid idea of what they are going to do.

Diamonds.

- These are collected by the player by running into them.
- They fall down when they have nothing to support them.
- A certain number or all of the diamonds need to be collected before the player can exit the level.
- Diamonds are able to kill the player and monsters when they fall on them. (Note: Falling will be defined more clearly when you implement the falling objects actually killing the player/monster).
- Diamonds will roll off other falling objects if there is enough space on either side.

Boulders.

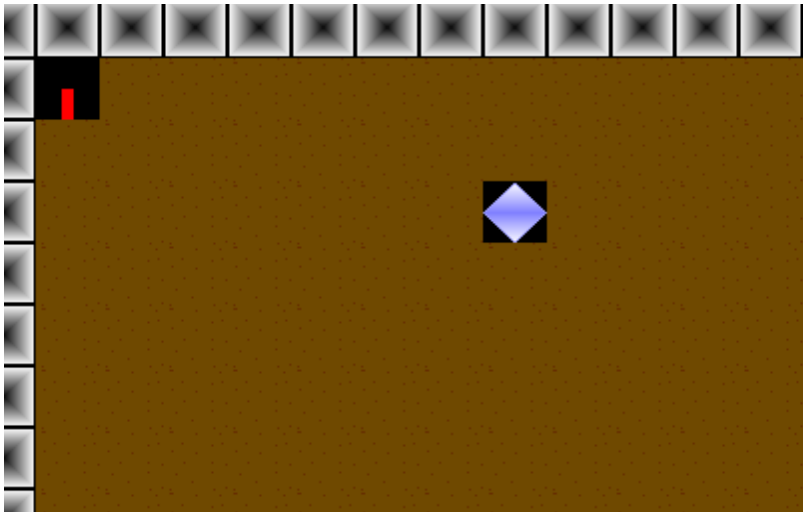
- These stop the player from moving into that space.
- They can be pushed left and right by the character into a clear space.
- They fall down when they have nothing to support them.
- Boulders are able to kill the player and monsters when they fall on them.
- Boulders will roll off other falling objects if there is enough space on either side.

The first, and seemingly the easiest thing to do is to make your character eat diamonds when he collides with them.

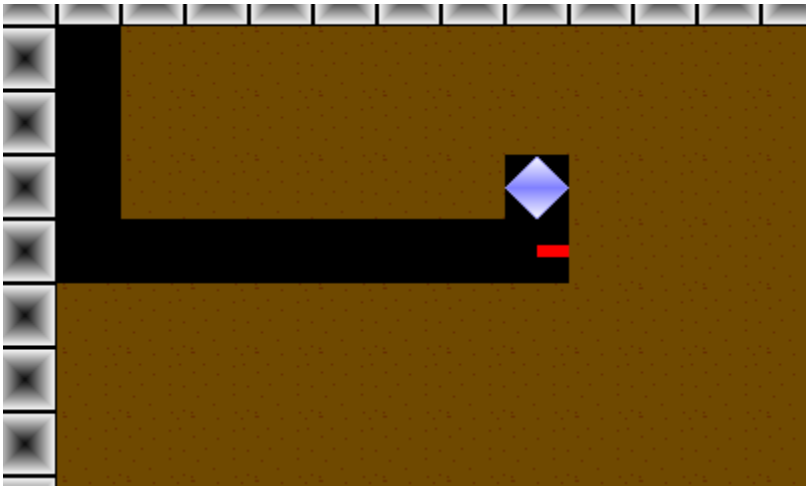
- Create an event that destroys the diamonds when the player steps on them.
- Make sure that you can eat these diamonds from all directions. Also your exit wall object should appear when you eat the diamond and you should be able to move into the exit wall object (but nothing will happen).

Now that your character can eat the diamonds, you need to work on the falling. First, do the simplest case: Make the objects fall down when they have nothing beneath them.

- In your test room, place a diamond above a square of dirt. Something like this



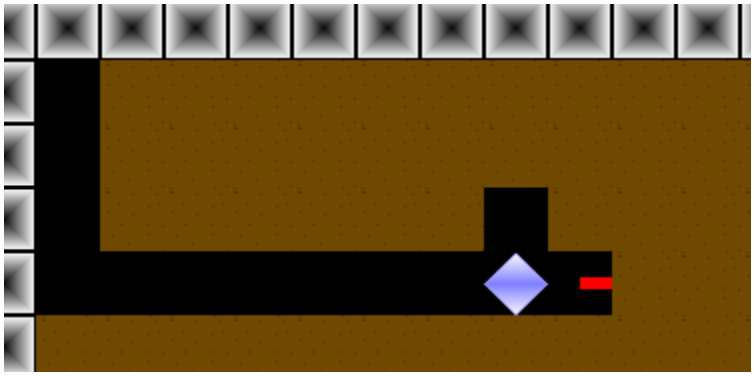
- Dig a hole underneath the diamond and stay there.



What would you expect if you moved out from underneath the diamond? What would happen if you were to move upwards into the diamond? What if you move down? This is the point at which you start to create the dynamics of your game – how things interact with each other.

- Set up an event that makes all falling objects fall down when there is nothing in the grid cell beneath them, use the 'If Any Object At' from the 'Collision' group in the toolbox. (Hint: Don't apply this to the diamond only. Look at the previous sentence more carefully.)
- Test to see that your diamond will fall down when you dig the dirt out from underneath it and move away.

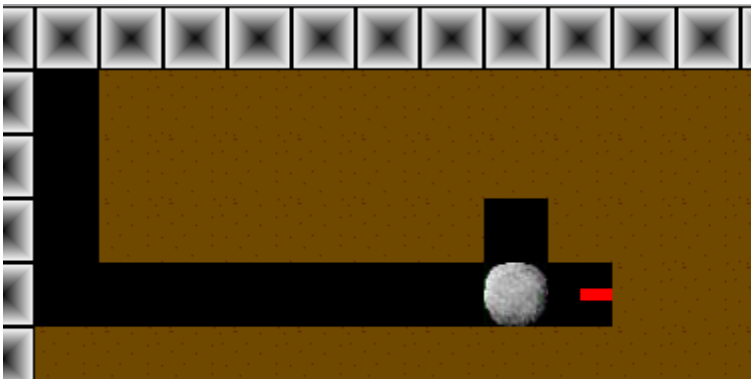
All going well, you'll get this effect.



This is the most basic implementation of falling. Your falling object will only fall straight down. It cannot roll left or right yet.

- Now try it with a boulder – for some reason too difficult to repair, the boulder is implemented as an object called objRock. Sorry about that – when we say boulder we mean rock. You might like to rename the rock object 'objboulder'.

If you have done everything correctly, you should get this.

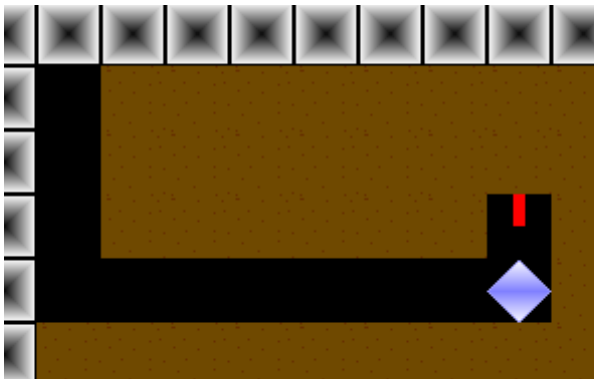


Since the boulder and the diamond are both children of the 'Falling Object' object, they take on the properties of the parent object. So whatever you tell the 'Falling Object' to do, the boulder and the diamond will do.

- Now, test with a diamond again, but this time, when you step underneath a diamond, walk up into it.
- If everything works – Rockford eats the diamond, then skip the next section.

SKIP THIS SECTION UNLESS YOU CANNOT EAT A DIAMOND FROM BELOW

Don't be surprised if you get this.



This is a nasty complexity that might arise – it doesn't always. If it happens to you we can give you some ideas. Try this:

- If you used a normal step event for making the falling object fall down, change that event to an end step event.
- Try the experiment again. Everything should work correctly.

The reason you weren't able to pick up the diamond by walking into it from below was because both the moving action of your character, and the falling action of the diamond were both happening in the same time step. Game Maker does all moves, calculates all collisions, then does all events. So your character moved into the square the diamond was in, colliding. Then the collision and the step event occur. The order in which they occur is effectively random. If the step event occurs first, the diamond sees that the area the player just came from is free, so it falls down, before the player can destroy it. By changing the step event for the Falling Parent to an 'End Step' event, you tell the game that you want to move the falling objects as the last thing. That means that your character will move onto that diamond and destroy it before the diamond gets a chance to move.

Resume here if your player eats diamonds correctly

The diamond is more or less taken care of, so now the laws for the boulder need to be put into place. As was stated earlier, the boulders may not be walked into by the character, but can be pushed left and right. WE will need to add some action to the code that is already in the Left movement event.

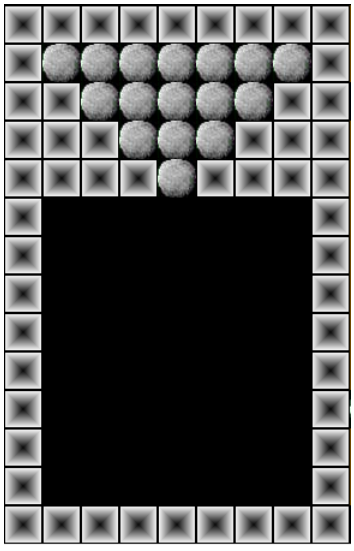
- For move left the full pseudo-code is:

```
IF there is a boulder to the left THEN
  IF there is a free space in the cell beyond the boulder THEN
      Destroy the object in the cell to the left
      Create a new boulder object in the cell beyond that
      Move the player to the left
      Set the sprite image to the left sprite image
ELSE
  IF there is free space to the left of the player THEN
      Move the player to the left
      Set the sprite image to the left sprite image
```

The pseudocode under the ELSE will be executed if there is not a boulder to the left.

- For the move right – something similar.
- Test thoroughly.

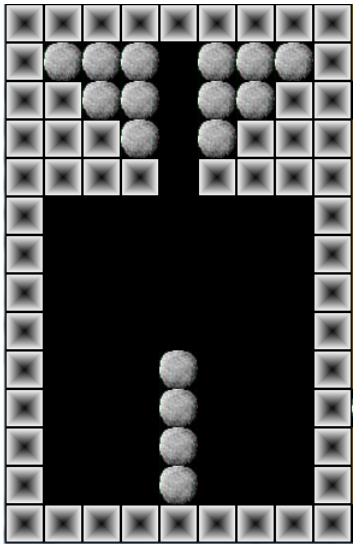
You cannot balance one boulder on top of another



Now that the individual properties of each falling object have been taken care of, the interaction between two of them needs to be done. In the original Boulderdash game, when you had one boulder sitting on top of another boulder or a diamond, the game would check to see if it could fall to the left and if it could, do so, and then do the same for the other side. In other words, one boulder (or diamond) would not balance on top of another. Here is the second room that I have created to test falling.

- Either make a copy of this, or create something similar in your own test room. Note that there is no dirt in the room – just a big empty space for boulders to fall into. Drag your room before Room0 so that it is the first room which is displayed.

When you run this room, the four boulders in the middle will fall down one after the other, and stack up in a line. You will probably see all this happen very fast. It looks much better if the boulders tumble slowly, and also gives you a chance to run away when you are beneath.



- Set the room speed to a low value. 10 steps per second is much more satisfactory than the default of 30. Try watching your room again.

This is not exactly what you want to happen. The boulders aren't rolling off each other. Each boulder must notice if it is standing on another boulder with a free space to the left or right and fall into that space. Consider first the possibility that there is a free space to the left.

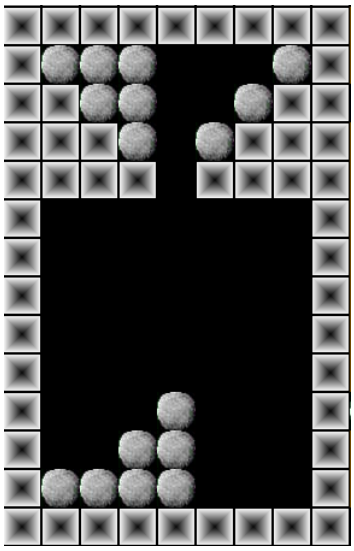
Add actions to the 'objFallingParent' object's Step or End Step event, after the check to see if the object can fall, based on the pseudo-code below:

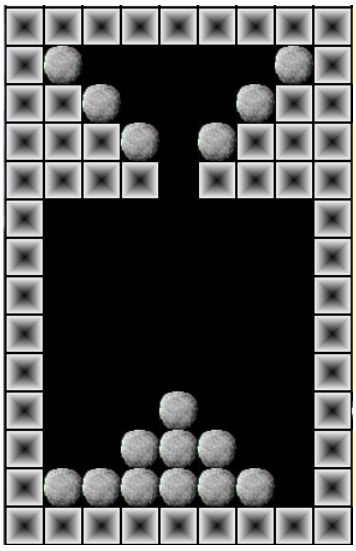
IF the object is sitting on another 'objFallingObject' THEN
  IF the grid cell to the left is free THEN
    IF the grid cell left and down is also free THEN
      Move the object one cell width (32 pixels) to the left.

It isn't necessary to also move it down – it will fall on the next step.

If you run your game after putting in the proper actions, you should get something looking like the left image below.

- Now implement the boulders falling in the opposite direction (to the right).





You should get something like the right image. You'll notice that if you watch at the top as the boulders fall, sometimes more than one slides left or right in a single step. This is a result of your actions doing what they're told, but not completely what was intended. Since fixing it would be quite a lot of trouble and the result acceptable, it can be left alone.

The next thing to be done is the check to see whether or not you're going to squash the player.

Step 4. I'm feeling a little flat today…

Getting exactly the right game play here requires a bit of work. When do boulders and diamonds squash the player. It's not as simple as checking to see if the player is below the boulder. In the original Boulderdash, the player could stand with a boulder on his head. The player was squashed only when the boulder had fallen from a higher position and landed on his head.

So, a further definition of a falling object needs to be created so as to implement it properly. You need to decide when the boulder is actually falling, and when it has come to rest. In the original game, falling objects were deemed to be falling only when they made their first step downwards. What is needed is a way to finding out whether or not the object is falling. This can be done with a variable commonly known as a flag. We will add a *flag* variable called 'falling' to each boulder.

Note: We don't need to do anything special to make a variable – just assign its value.
A flag is usually a true/false variable (known as a boolean variable) that tells the user/game that a statement is true or false. The statement we will be checking is "Is the falling object falling?"

When you set the falling variable to false, the object is not falling. When it is set to true, it is falling). In the Create event for the falling parent object assign a variable called varFalling to false.

The first step to getting squishing right is to set the variable 'falling'. The second step is to sort out the different cases of falling for boulders and diamonds. At present we have three:

- If there is nothing below a boulder (or diamond) it falls.
- If there is a boulder or diamond below and clear spaces to the left it rolls left.
- If there is a boulder or diamond below and clear spaces to the right it rolls right.

To these cases we must add a fourth:

- If there is a player (and later we will want to add 'or monster' here) below, and the last step was a fall (falling is true), it falls (squashing the player).

This can all be done by making these changes.

- Add a test for the new fourth case before the tests for the other cases.
  i.e.: If falling is true and then if there is a player directly below then fall.
- When an object falls directly down, set the variable falling to true.
- If the object doesn't fall directly down, set falling to false.

- Try walking under the boulders and diamonds.

Now that you have your falling all fixed, you can now create a collision event with a set of actions to destroy your player when he gets squished. In the original game, when the player was killed, everything in a radius 1 circle around the player was also killed. So that's every one of the 8 points on the compass around you.

First you need to destroy your player when he's squished.

- Destroy the character when the falling object collides with the player.

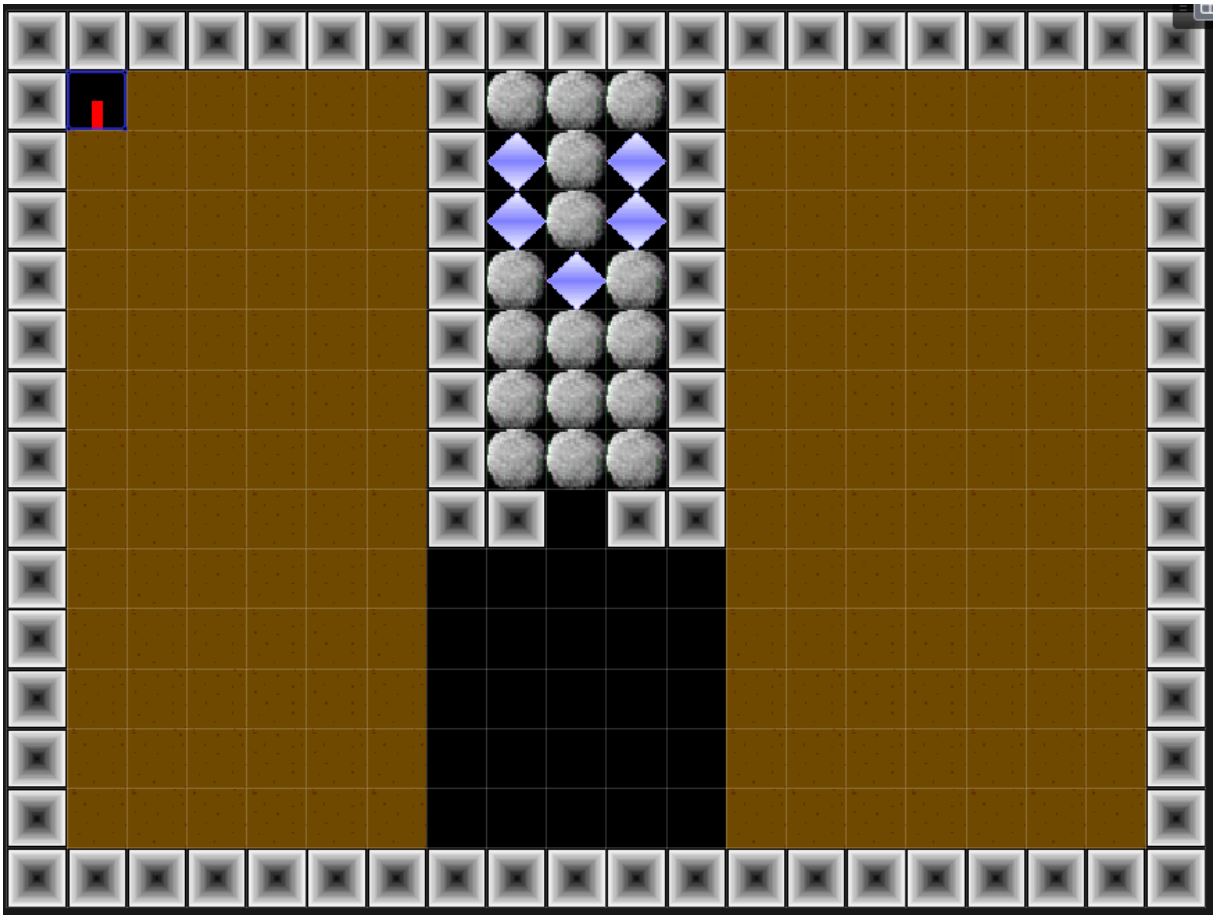
Now that you have the event that destroys the character, you need to do the extra actions to explode everything around.

- Load up the sprite "Explosion.gif" and remove the background and set the animation speed to 10. Create an object using that sprite. Do not parent it with anything.
- After the player is destroyed, create 9 instances of the explosion in the 9 areas in and around your player. E.g.

x: -32 y: -32	x: 0 y: -32	x: 32 y:-32
x: -32 y: 0	x: 0 Player y: 0	x: 32 y: 0
x: -32 y: 32	x: 0 y: 32	x: 32 y: 32

- Make the explosion destroy everything it collides with that is not a titanium wall (parenting is useful here). Even boulders and diamonds are destroyed by explosions.
- After the animation ends, destroy the explosion so that you're just left with a 3x3 square of nothing.
- If you don't like the way the explosions still occur above the walls, set both the wall and the exit's depth to -10, so that they will always be drawn on top of the explosions.

Now that you have the basics of the game working, create the puzzle on the following page, and see if you can complete it. Put the exit in one of the sides, or the top walls.



And before you ask the tutors and friends, yes it can be completed. I wouldn't give it to you if it couldn't. ☺

One last thing that needs to be added is the ability to restart the game if you become trapped or die. For this action to always occur, you need to place this event in a single object that will always exist in each level. This object would be the exit, since it can't be destroyed in any way when the game is running.

- In the 'objPlayer' object place an event that activates when the 'r' key is pressed.
- In that event, restart the level.

Session 4: REVIEW PAGE **Name:** MIN SOE HTUT

Questions: Complete the following questions and be prepared to demonstrate techniques to your demonstrator. (You can then have your work verified.)

1. Why don't we have key release events? If we did, what might they be used for?

it may be use for when release the key the player will stop moving..

2. What is the 'Else' action used for?

after the if statement when the if statement does not meet the requirement it will do the else statement

```
if
  {
  }
else
  {
  }
```

3. Why must the test for a boulder falling on a player be done before the test for falling into a free space?

By first checking if there is a player below, you ensure that the player is squashed only when a boulder has descended and is in the act of falling directly onto the player's position. If you checked for a free space first, it could lead to scenarios where the boulder falls into an empty space without harming the player, even if it lands on the player's head during the same step.

`

Verification: To assess your competency of this material your demonstrator will verify that you have completed the above questions, and can do the following:

1) Create a new variable and set it to 10
2) Check that the variable is less than or equal to 10

Games – Pacman to Boulderdash – Session Five

More on Boulderdash: Puzzle

Make

There is just one thing to do in this session. Make a room for Boulderdash. Make it interesting. The kinds of puzzles found in the original version were:

- It might be hard to reach some diamonds. Perhaps the player needs to cleverly drop or push some boulders to make a path, without getting killed in the process.

- If might be hard to get to the exit after collecting diamonds. In some frames the exit was close to the starting point. After collecting diamonds you might have made it impossible to get back.

- Dealing with monsters. Can you trap a monster by virtue of its simplistic movement strategy? Can you drop a boulder on a monster?

Guidelines

Put some pieces on the board and experiment. You will find that a reasonably full screen (lots of walls and boulders) is likely to offer interesting play.

Test

Try your frame on someone else. How easily did they solve it?

Creepy crawlies. (OPTIONAL).

No game would be complete without a little bit of monster-ism, and Boulderdash was no exception, back in the days of its popularity. Fireflies, rats, and amoeba stood in your way to escaping the mines with your riches. And you're going to program one of them!

Remember how you saw the left wall hugger in the pacman clone? That's what you're going to create in this as well. You already have the firefly object created, so all you have to do is give it the actions to make it move.

Improving the graphics. (OPTIONAL).

You will notice that even though this is a game made using quite a powerful creation tool, that the graphics are rather sub par. The player is represented by a red line pointing in the direction he is facing. The monster (if you make it) only has one way it's facing.

Create or find some new graphics to put into the game to give it a more real feel to it. You could even try and recreate some of the original graphics of Boulderdash.

Session 5: REVIEW PAGE **Name: _____**

Questions: Complete the following questions and be prepared to demonstrate techniques to your demonstrator. (You can then have your work verified.)

There are no questions. Your demonstrator will try to play your level.

Verification: Your demonstrator will ask to see your game at this stage.