A powerful new game maker. Build your games in the world's best editor. No prior experience in programming is necessary to use our event system. Be up and running in minutes. HTML5 is the future of the web. **Construct 2** makes HTML5 games that run in a browser. Run your games on Mac, iPhone, iPad, Windows, Linux and more! Just like Flash, but no plugins necessary.
Physics.................................................................................................................. 31
  How to add Physics .................................................................................................. 31
    Gravity ................................................................................................................. 31
    Making the scenery .............................................................................................. 31
  The other physics properties .................................................................................. 32
  General physics tips ................................................................................................ 33
    Performance ......................................................................................................... 33
    Stability .............................................................................................................. 33
    Manual movement & other behaviors .................................................................. 34
  Conclusion ................................................................................................................ 34

More Physics: Forces, impulses, torque and joints................................................. 35
  What are forces, impulses and joints? ................................................................. 35
  How to use forces, impulses, torque and joints .................................................. 35
  Forces ................................................................................................................... 36
    Point of application .............................................................................................. 37
  Impulses ................................................................................................................ 37
  Torque ................................................................................................................... 38
  Joints ..................................................................................................................... 38
    Revolute joint ...................................................................................................... 38
    Distance joint ...................................................................................................... 38
  Don't forget... ........................................................................................................ 39
  Conclusion ................................................................................................................ 39

Delta-time and framerate independence ............................................................... 40
  The $dt$ system expression .................................................................................... 40
    How to use $dt$ ...................................................................................................... 40
    Use $dt$ everywhere ............................................................................................. 41
    Behaviors already use $dt$ .................................................................................. 41
  Timescaling ............................................................................................................. 41
  Pausing .................................................................................................................. 41

Other kinds of movement ....................................................................................... 42
  Rotation ................................................................................................................. 42
  Acceleration .......................................................................................................... 42

Advanced considerations ........................................................................................ 42
Minimum framerate ................................................................. 42
Random variation ........................................................................... 42
Object time scales ......................................................................... 43
Conclusion ...................................................................................... 43

Upload your game to Dropbox ....................................................... 44
  Get a Dropbox account ............................................................... 44
  Export your project ....................................................................... 44
  Log in to Dropbox ........................................................................ 45
  Uploading the game files ............................................................. 45
  Getting a public link ..................................................................... 46
  Sharing the link ........................................................................... 46
  Using the desktop application ..................................................... 46

Questions and Answers ................................................................ 47
  Q: How do I move a sprite in a sine wave pattern? ...................... 47
  Q: How do I use a single image tile sheet? .................................. 47
  Q: How do I loop an animation? .................................................. 47
Now you're set up, launch Construct 2. Click the File button, and select New.

In the New Project dialog, you don't need to change anything. Just click Create project. Construct 2 will keep the entire project in a single .capx file for us. You should now be looking at an empty layout - the design view where you create and position objects. Think of a layout like a game level or menu screen. In other tools, this might have been called a room, scene or frame.

**Inserting objects**

**Tiled Background**

The first thing we want is a repeating background tile. The Tiled Background object can do this for us. First, here's your background texture - right click it and save it to your computer somewhere:

Now, double click a space in the layout to insert a new object. (Later, if it's full, you can also right-click and select Insert new object.) Once the Insert new object dialog appears, double click the Tiled Background object to insert it.
A crosshair will appear for you to indicate where to place the object. Click somewhere near the middle of the layout. The texture editor now opens, for you to enter the texture to tile. Let's import the tile image you saved earlier. Click the folder icon to load a texture from disk, find where you downloaded the file to, and select it.

Close the texture editor by clicking the X in the top right. If you're prompted, make sure you save!

Now you should see your tiled background object in the layout. Let's resize it to cover the entire layout. Make sure it's selected, then the Properties Bar on the left should show all the settings for the object, including its size and position. Set its position to 0, 0 (the top left of the layout), and its size to 1280, 1024 (the size of the layout).
Let's survey our work. Hold **control** and scroll the **mouse wheel down** to zoom out. Alternatively, click **view** - zoom out a couple of times. You can also hold space, or the middle mouse button, to pan around. Neat, huh? Your tiled background should cover the entire layout now:

Hit control+0 or click **view** - zoom to 100% to return to 1:1 view.

*(If you're impatient like me, click the little 'run' icon in the window title bar - a browser should pop up showing your tiled layout! Woo!)*

**Adding a layer**

Okay, now we want to add some more objects. However, we're going to keep accidentally selecting the tiled background unless we **lock** it, making it unselectable. Let's use the layering system to do this.

Layouts can consist of multiple **layers**, which you can use to group objects. Imagine layers like sheets of glass stacked on top of each other, with objects painted on each sheet. It allows you to easily
arrange which objects appear on top of others, and layers can be hidden, locked, have parallax effects applied, and more. For example, in this game, we want everything to display above the tiled background, so we can make another layer on top for our other objects.

To manage layers, click the **Layers tab**, which usually is next to the **Project bar**:

You should see *Layer 0* in the list (Construct 2 counts starting from zero, since it works better like that in programming). Click the pencil icon and **rename it** to *Background*, since it's our background layer. Now click the green 'add' icon to add a new layer for our other objects. Let's call that one *Main*. Finally, if you click the little padlock icon next to *Background*, it will become **locked**. That means you won't be able to select anything on it. That's quite convenient for our tiled background, which is easy to accidentally select and won't need to be touched again. However, if you need to make changes, you can just click the padlock again to unlock.

The checkboxes also allow you to hide layers in the editor, but we don't need that right now. Your layers bar should now look like this:

Now, **make sure the 'Main' layer is selected in the layers bar**. This is important - the selected layer is the **active** layer. All new inserted objects are inserted to the **active** layer, so if it's not selected, we'll be accidentally inserting to the wrong layer. The active layer is shown in the status bar, and also appears in a tooltip when placing a new object - it's worth keeping an eye on.
Add the input objects

Turn your attention back to the layout. Double click to insert another new object. This time, select the Mouse object, since we'll need mouse input. Do the same again for the Keyboard object.

Note: these objects don't need placing in a layout. They are hidden, and automatically work project-wide. Now all layouts in our project can accept mouse and keyboard input.

The game objects

It's time to insert our game objects! Here are your textures - save them all to disk like before.

Player:

Monster:

Bullet: 

and Explosion:

For each of these objects, we will be using a sprite object. It simply displays a texture, which you can move about, rotate and resize. Games are generally composed mostly out of sprite objects. Let's insert each of the above four objects as sprite objects. The process is similar to inserting the Tiled Background:

1. **Double click** to insert a new object
2. **Double click** the 'Sprite' object.
3. When the mouse turns to a crosshair, click somewhere in the layout. The tooltip should be 'Main'. (Remember this is the active layout.)
4. The texture editor pops up. Click the open icon, and **load one of the four textures**.
5. **Close** the texture editor, saving your changes. You should now see the object in the layout!
Note: another quick way to insert sprite objects is to drag and drop the image file from Windows in to the layout area. Construct 2 will create a Sprite with that texture for you.

Move the bullet and explosion sprites to somewhere off the edge of the layout - we don't want to see them when the game starts.

These objects will be called Sprite, Sprite2, Sprite3 and Sprite4. That's not very useful - things will quickly get confusing like this. Rename them to Player, Monster, Bullet and Explosion as appropriate. You can do it by selecting the object, then changing the Name property in the properties bar:

![Image of Object Type Properties]

### Adding behaviors

Behaviors are pre-packaged functionality in Construct 2. For example, you can add a Platform behavior to an object, and the Solid behavior to the floor, and you instantly can jump around like a platformer. You can do the same in events, but it takes longer, and there's no point anyway if the behavior is already good enough! So let's have a look at which behaviors we can use. Amongst others, Construct 2 has these behaviors;

- **8 Direction movement.** This lets you move an object around with the arrow keys. It will do nicely for the player's movement.
- **Bullet movement.** This simply moves an object forwards at its current angle. It'll work great for the player's bullets. Despite the name, it'll also work nicely to move the monsters around - since all the movement does is move objects forwards at some speed.
- **Scroll to.** This makes the screen follow an object as it moves around (also known as scrolling). This will be useful on the player.
- **Bound to layout.** This will stop an object leaving the layout area. This will also be useful on the player, so they can't wander off outside the game area!
- **Destroy outside layout.** Instead of stopping an object leaving the layout area, this destroys it if it does. It's useful for our bullets. Without it, bullets would fly off the screen forever, always taking a little bit of memory and processing power. Instead, we should destroy the bullets once they've left the layout.
- **Fade.** This gradually makes an object fade out, which we will use on the explosions.

Let's add these behaviors to the objects that need them.

### How to add a behavior

Let's add the **8 direction movement** behavior to the player. Click the player to select it. In the properties bar, notice the Behaviors category. Click Add / Edit there. The Behaviors dialog for the player will open.
Click the green 'add behavior' icon in the behaviors dialog. Double-click the **8 direction movement** to add it.

Do the same again and this time add the **Scroll To** behavior, to make the screen follow the player, and also the **Bound to layout** behavior, to keep them inside the layout. The behaviors dialog should now look like this:
Close the behaviors dialog. Hit Run to try the game!

Hopefully you have a HTML5 compatible browser installed. Otherwise, be sure to get the latest version of Firefox or Chrome, or Internet Explorer 9 if you're on Vista and up. Once you have the game running, notice you can already move around with the arrow keys, and the screen follows the player! You also can't walk outside the layout area, thanks to the Bound to Layout behavior. This is what behaviors are good for - quickly adding common bits of functionality. We'll be using the event system soon to add customised functionality.

**Adding the other behaviors**

We can add behaviors to the other objects by the same method - select it, click *Add / Edit* to open the behaviors dialog, and add some behaviors. Let's add those other behaviors:

- Add the **Bullet movement** and **Destroy outside layout** to the **Bullet** object (no surprises there)
- Add the **Bullet movement** to the **Monster** object (because it just moves forwards as well)
- Add the **Fade** behavior to the **Explosion** object (so it gradually disappears after appearing). By default the Fade behavior also destroys the object after it has faded out, which also saves us having to worry about invisible Explosion objects clogging up the game.

If you run the game, you might notice the only thing different is any monsters you can see suddenly shoot off rather quickly. Let's slow them down to a leisurely pace. Select the **Monster** object. Notice how since we added a behavior, some extra properties have appeared in the properties bar:

This allows us to tweak how behaviors work. Change the speed from 400 to 80 (this is in pixels travelled per second).

Similarly, change the **Bullet object's** speed to 600, and the **Explosion** object's Fade behavior's **Fade out time** to 0.5 (that's half a second).
Create some more monsters

Holding control, click and drag the **Monster** object. You'll notice it spawns another *instance*. This is simply another object of the Monster *object type*.

Object types are essentially 'classes' of objects. In the event system, you mainly deal with object types. For example, you might make an event that says "Bullet collides with Monster". This actually means "*Any instance of the Bullet object type collides with any instance of the Monster object type*" - as opposed to having to make a separate event for each and every monster. With Sprites, all instances of an object type also share the same texture. This is great for efficiency - when players play your game online, rather than having to download 8 monster textures for 8 monsters, they only need to download one monster texture and Construct 2 repeats it 8 times. We'll cover more on *object types vs. instances* later. For now, a good example to think about is different types of enemy are different object types, then the actual enemies themselves (which there might be several of) are instances of those object types.

Using control + drag, **create 7 or 8 new monsters**. Don't place any too close to the player, or they might die straight away! You can zoom out with control + mouse wheel down if it helps, and spread them over the whole layout. You should end up with something a bit like this.

Now it's time to add our custom functionality via Construct 2's visual method of programming - the *event system*.

**Events**

First, click the **Event sheet 1** tab at the top to switch to the *Event sheet editor*. A list of events is called an *Event sheet*, and you can have different event sheets for different parts of your game, or for organisation. Event sheets can also "include" other event sheets, allowing you to reuse events on multiple levels for example, but we won't need that right now.
About events

As the text in the empty sheet indicates, Construct 2 runs everything in the event sheet once per tick. Most monitors update their display 60 times per second, so Construct 2 will try to match that for the smoothest display. This means the event sheet is usually run 60 times per second, each time followed by redrawing the screen. That's what a tick is - one unit of "run the events then draw the screen".

Events run top-to-bottom, so events at the top of the event sheet are run first.

Conditions, actions and sub-events

Events consist of conditions, which test if certain criteria are met, e.g. "Is spacebar down?". If all these conditions are met, the event's actions are all run, e.g. "Create a bullet object". After the actions have run, any sub-events are also run - these can then test more conditions, then run more actions, then more sub-events, and so on. Using this system, we can build sophisticated functionality for our games and apps. We won't need sub-events in this tutorial, though.

Let's go over that again. In short, an event basically runs like this:

Are all conditions met?

--- Yes: run all the event's actions.
--- No: go to next event (not including any sub-events).

That's a bit of an oversimplification. Construct 2 provides a lot of event features to cover lots of different things you might need to do. However, for now, that's a good way to think about it.

Your first event

We want to make the player always look at the mouse. It will look like this when we're done:

<table>
<thead>
<tr>
<th>1</th>
<th>System</th>
<th>Every tick</th>
<th>Player</th>
<th>Set angle toward (Mouse.X, Mouse.Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add action</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember a tick runs every time the screen is drawn, so if we make the player face the mouse every tick, they'll always appear to be facing the mouse.

Let's start making this event. Double-click a space in the event sheet. This will prompt us to add a condition for the new event.
Different objects have different conditions and actions depending on what they can do. There’s also the **System object**, which represents Construct 2’s built-in functionality. **Double-click** the System object as shown. The dialog will then list all of the System object's conditions:

**Double-click** the *Every tick* condition to insert it. The dialog will close and the event is created, with no actions. It should now look like this:
Now we want to add an action to make the player look at the mouse. Click the Add action link to the right of the event. (Make sure you get the Add action link, not the Add event link underneath it which will add a whole different event again.) The Add Action dialog will appear:

![Add Action Dialog](image)

As with adding an event, we have our same list of objects to choose from, but this time for adding an action. Try not to get confused between adding conditions and adding actions! As shown, double-click the Player object, for it is the player we want to look at the mouse. The list of actions available in the Player object appears:

![Add Action Dialog](image)
Notice how the player's 8-direction movement behavior has its own actions. We don't need to worry about that for now, though.

Rather than set the player's angle to a number of degrees, it's convenient to use the **Set angle towards position** action. This will automatically calculate the angle from the player to the given X and Y co-ordinate, then set the object's angle to that. **Double-click** the **Set angle towards position** action.

Construct 2 now needs to know the X and Y co-ordinate to point the player at:

![Parameters for Player: Set angle toward position](image)

These are called the **parameters** of the action. Conditions can have parameters too, but **Every tick** doesn't need any.

We want to set the angle towards the mouse position. The Mouse object can provide this. Enter **Mouse.X** for **X**, and **Mouse.Y** for **Y**. These are called **expressions**. They're like sums that are calculated. For example, you could also enter **Mouse.X + 100** or **sin(Mouse.Y)** (although those particular examples might not be very useful!). This way you can use any data from any object, or any calculation, to work out parameters in actions and conditions. It's very powerful, and a sort of hidden source of much of Construct 2's flexibility.

You might be wondering how you'd remember all the possible expressions you could enter. Luckily, there's the "object panel" which you should see floating above it. By default, it's faded out so it doesn't distract you.

![Objects with expressions](image)

Hover the mouse over it, or click on it, and it'll become fully visible. This serves as a sort of dictionary of all the expressions you can use, with descriptions, to help you remember. If you double-click an
object, you'll see all its expressions listed. If you double-click an expression, it will also insert it for you, saving you from having to type it out.

Anyway, click **Done** on the parameters dialog. The action is added! As you saw before, it should look like this:

```
1. System > Every tick
   Player > Set angle towards (Mouse.X, Mouse.Y)
```

There's your first event! Try running the game, and the player should now be able to move around as before, but always facing the mouse. This is our first bit of custom functionality.

## Adding game functionality

If each event is described in as much detail as before, it's going to be quite a long tutorial. Let's make the description a little briefer for the next events. Remember, the steps to add a condition or action are:

1. Double-click to insert a new event, or click an **Add action** link to add an action.
2. Double-click the object the condition/action is in.
3. Double-click the condition/action you want.
4. Enter parameters, if any are needed.

From now on, events will be described as the object, followed by the condition/action, followed by any parameters. For example, the event we have just inserted could be written:

Add condition **System** -> **Every tick**  
Add action **Player** -> **Set angle towards position** -> **X: Mouse.X, Y: Mouse.Y**  

### Get the player to shoot

When the player clicks, they should shoot a bullet. This can be done with the **Spawn an object** action in **Player**, which creates a new instance of an object at the same position and angle. The **Bullet movement** we added earlier will then make it fly out forwards. Make the following event:

**Condition:** **Mouse** -> **On click** -> **Left clicked** (the default)  
**Action:** **Player** -> **Spawn another object** -> For **Object**, choose the **Bullet** object. For **Layer**, put 1 (the "Main" layer is layer 1 - remember Construct 2 counts from zero). Leave **Image point** as 0.

Your event should now look like this:

```
3. Mouse > Left button Clicked
   Player > Spawn Bullet on layer 1 (image point 0)
```

If you run the game, you'll notice the bullets shoot from the middle of the player, rather than from the end of the gun. Let's fix that by placing an **image point** at the end of the gun. (An image point is just a position on an image that you can spawn objects from.)

**Right-click** the player in the project or object bar and select **Edit animations**.
The image editor for the player reappears. Click the origin and image points tool:

...and the image points dialog opens up:

Notice the object origin appears as a red spot. That's the "hotspot" or "pivot point" of the object. If you rotate the object, it spins around the origin. We want to add another image point to represent the gun, so click the green *add* button. A blue point appears - that's our new image point. Left-click at the end of the player's gun to place the image point there:

Close the image editor. Double-click the *Spawn an object* action we added earlier, and change the *Image point* to 1. (The origin is always the first image point, and remember Construct 2 counts from zero.) The event should now look like below - note it says *Image point 1* now:
Run the game. The bullets now shoot from the end of your gun! The bullets don't do anything yet, though. Hopefully, however, you'll start to realise that once you get to grips with the event system, you can put functionality together very quickly.

Let's make the bullets kill monsters. Add the following event:

**Condition:** Bullet -> On collision with another object -> pick Monster.
**Action:** Monster -> Destroy
**Action:** Bullet -> Spawn another object -> Explosion, layer 1
**Action:** Bullet -> Destroy

**The explosion effect**

Run the game, and try shooting a monster. Oops, the explosion has that big black border!

You might have predicted it'd look like that from the start, and wondered if our game was really going to end up like that! Don't worry, it won't. **Click the Explosion object** in either the Object bar in the bottom right, or the Project bar (which was tabbed with the layers bar). Its properties appear in the properties bar on the left. At the bottom, set its **Effect** property to **Additive**. Now try the game again.

Why does this work? Without going in to the nuts and bolts, ordinary images are *pasted on top* of the screen. With the additive effect, each pixel is instead *added* (as in, summed) with the background pixel behind it. Black is a zero pixel value, so nothing gets added - you don't see the black background. Brighter colors add more, so appear more strongly. It's great for explosions and lighting effects.
Making monsters a little smarter

Right now the monsters just wander off the layout to the right. Let's make them a bit more interesting. First of all, let's start them at a random angle.

Condition: System -> On start of Layout
Action: Monster -> Set angle -> random(360)

They will still wander off forever when they leave the layout, never to be seen again. Let's keep them inside. What we'll do is point them back at the player when they leave the layout. This does two things: they always stay within the layout, and if the player stands still, monsters come right for them!

Condition: Monster -> Is outside layout
Action: Monster -> Set angle toward position -> For X, Player.X - for Y, Player.Y.

Run the game. If you hang around for a while, you'll notice the monsters stay around the layout too, and they're going in all kinds of directions. It's hardly AI, but it'll do!

Now, suppose we want to have to shoot a monster five times before it dies, rather than instant death like it is at the moment. How do we do that? If we only store one "Health" counter, then once we've hit a monster five times, all the monsters will die. Instead, we need each monster to remember its own health. We can do that with instance variables.

Instance variables

Instance variables allow each monster to store its own health value. A variable is simply a value that can change (or vary), and they are stored separately for each instance, hence the name instance variable.

Let's add a health instance variable to our monster. Click the monster in the project bar or object bar. Alternatively, you can switch back to the layout and select a monster object. This will show the monster's properties in the properties bar. Click Add/edit by Edit variables.
The Instance Variables dialog appears. It looks similar to the Behaviors dialog we saw earlier, but instead allows you to add and change instance variables for the object. Click the green **Add** button to add a new one.

![New instance variable dialog]

In the dialog that pops up, type **health** for the name, leave **Type** as **Number**, and for **Initial value** enter **5** (as shown). This starts every monster on 5 health. When they get hit we'll subtract 1 from the health, and then when health is zero we'll destroy the object.

Once you're done click OK. Notice the variable now appears in the instance variables dialog and also in the properties for the monster as well. (You can quickly change initial values in the properties bar, but to add or remove variables you'll need to click the **Add / Edit** link.)

![Instance variables properties]

### Changing the events

Switch back to the event sheet. Right now, we're destroying monsters as soon as the bullet hits them. Let's change that to subtract 1 from its health.

Find the event that reads: **Bullet - on collision with Monster**. Notice we've got a "destroy monster" action. Let's replace that with "subtract 1 from health". Right click the "destroy monster" action and click **Replace**.
The same dialog appears as if we were inserting a new action, but this time it'll replace the action we clicked instead. Choose Monster -> Subtract from (in the Instance variables category) -> Instance variable "health", and enter 1 for Value. Click Done. The action should now appear like this:

Now when we shoot monsters they lose 1 health and the bullet explodes, but we haven't made an event to kill monsters when their health reaches zero. Add another event:

Condition: Monster -> Compare instance variable -> Health, Less or equal, 0
Action: Monster -> Spawn another object -> Explosion, layer 1
Action: Monster -> Destroy

Why "less or equal 0" rather than "equals 0"? Suppose we added another more powerful weapon which subtracted 2 from health. As you shot a monster, its health would go 5, 3, 1, -1, -3... notice at no point was its health directly equal to zero, so it'd never die! Therefore, it's good practice to use "less or equal" to test if something's health has run out.

Run the game. You now have to hit monsters five times to kill them!

**Keeping score**

Let's have a score so the player knows how well they've done. We'll need another variable for this. You might think "let's put the score as one of the player's instance variables!". That's not a bad first idea, but remember the value is stored "in" the object. If there are no instances, there are no variables either! So if we destroy the player, we can no longer tell what their score was, because it was destroyed with the player.

Instead, we can use a **global variable**. Like an instance variable, a global variable (or just "global") can store text or a number. However, there's only ever one value stored, and it's also available to the entire game across all layouts - convenient if we were to add other levels.

Right-click the space at the bottom of the event sheet, and select Add global variable.
Enter **Score** as the name. The other field defaults are OK, it'll make it a number starting at 0.

Now the global variable appears as a line in the event sheet. It's in this event sheet, but it can be accessed from any event sheet in any layout.

![New global variable dialog](image)

Now the player has a score, which increases by 1 for every monster they kill - but they can't see their score! Let's show it to them with a text object.

**Note**: there are also local variables which can only be accessed by a smaller "scope" of events, but we don't need to worry about that right now.

Let's give the player a point for killing a monster. In our "Monster: health less or equal 0" event (when a monster dies), click **Add action**, and select **System -> Add to** (under Global & local variables) -> **Score**, value 1. Now the event should look like this:

![Event sheet](image)

Now the player has a score, which increases by 1 for every monster they kill - but they can't see their score! Let's show it to them with a text object.
Creating a heads-up display (HUD)

A heads-up display (aka HUD) is the interface that shows the player's health, score and other information in-game. Let's make a really simple HUD out of a text object.

The HUD always stays the same place on the screen. If we have some interface objects, we don't want them to scroll away as the player walks around - they should stay on the screen. By default, layers scroll. To keep them on the screen, we can use the layer Parallax setting. Parallax allows different layers to scroll at different rates for a sort of semi-3D effect. If we set the parallax to zero, though, the layer won't scroll at all - ideal for a HUD.

Go back to the layers bar we used earlier. Add a new layer called HUD. Make sure it's at the top, and selected (remember this makes it the active layer). The Properties bar should now be displaying its properties. Set the Parallax property to 0, 0 (that's zero on both the X and Y axes).

Double-click a space to insert another object. This time pick the Text object. Place it in the top left corner of the layout. It's going to be hard to see if it's black, so in the properties bar, make it bold, italic, yellow, and choose a slightly larger font size. Resize it wide enough to fit a reasonable amount of text. It should look something like this:

Switch back to the event sheet. Let's keep the text updated with the player's score. In the Every tick event we added earlier, add the action Text -> Set text.

Using the & operator, we can convert a number to text and join it to another text string. So for the text, enter:

"Score: " & Score

The first part ("Score: ") means the text will always begin with the phrase Score:. The second part (Score) is the actual value of the Score global variable. The & joins them together in to one piece of text.

Run the game, and shoot some monsters. Your score is displayed, and it stays at the same place on the screen!

Finishing touches

We're nearly done. Let's add some final touches.

Firstly, let's have some monsters regularly spawning, otherwise once you've shot all the monsters there's nothing left to do. We'll create a new monster every 3 seconds. Add a new event:

Condition: System -> Every X seconds -> 3
Action: System -> Create object -> Monster, layer 1, 1400 (for X), random(1024) (for Y)
1400 is an X co-ordinate just off the right edge of the layout, and \textit{random(1024)} is a random Y co-ordinate the height of the layout.

Finally, let's have ghosts kill the player.

\textit{Condition: Monster \texttt{--> On collision with another object \texttt{-}} \textit{Player}}

\textit{Action: Player \texttt{--> Destroy}}

\section*{Conclusion}

Congratulations, you've made your first HTML5 game in Construct 2! If you have a server and want to show off your work, click \textbf{Export} in the File menu. Construct can save all the project files to a folder on your computer, which you can upload or integrate to a web page. If you don't have your own server, you can \underline{share your games on Dropbox}.

You've learnt some important basics about Construct2: inserting objects, using layers, behaviors, events and more. Hopefully this should leave you well prepared to learn more about Construct 2! Try exploring its features and see what it can do for you.

---oOo---

\section*{Basic Loops and Arrays}

\textbf{Created by cow_trix}


In this tutorial we'll look at the basics of arrays and loops in Construct 2. These two elements are quite powerful and you can make some very cool things with them, especially when combined.

\section*{Loops In Theory}

A loop is an event that is executed a number of times, or 'iterations', every \texttt{tick}. There are two types of loops that you need to know about: the \textit{While} loop and the \textit{For} loop.

A \textit{while} loop executes an instruction while a condition is true, e.g.

\begin{enumerate}
  \item While (you are breathing)
  \item You are alive!
\end{enumerate}

As you can see, a while loop is useful when you don't know how many iterations you're going to have - after all, you don't know how long you're going to be breathing, but while you are, you'll be alive.

A \textit{for} loop can be considered a subcategory of a while loop. It is used when you know how many iterations you want, e.g.
1: For 0 to 9
2: Print “Hello!”

This will print “Hello!” 10 times. It’s important to remember Construct 2 (and a lot of programming languages) are zero-based as a rule (however, this can get confusing when we get to Arrays, but don’t worry about that just yet).

**Loops in Construct 2**

Most loops in Construct are *system conditions*. You can insert one by creating a new event, selecting ‘System’ and then a condition under the heading ‘Loops’.

![Loops in Construct 2](image)

**For**

After creating a For loop, you’ll be prompted for 3 fields:

*Name* - the name of the loop. Try to give it something descriptive, so it gives you an idea of what it’s counting.

*Start Index* - This is the number the loop starts counting at.

*End Index* - This is the number the loop finishes at.

Keep in mind that both indexes are *inclusive*. Also, if the start index is greater than the end index, the loop will not execute.

A *while* loop can be made by putting a variable in either of these index fields like so:
**For Each**

This type of loop will execute once for each instance of an object. It is important to note that events included in this loop relating to the object specified will pick to that object. E.g. the following events will create a text object for each 'Button' sprite, move it to the button's location, and set the text to the Button's "ButtonText" variable.

It's important to note that any 'For Each' event will pick to the object it is currently counting.

**Repeat**

The repeat event just repeats a series of events a set number of times. Note that this number can be a variable (and therefore adjusted at runtime).

**Arrays In Theory**

If you've ever done Euclidean geometry, you've already encountered a form of arrays in the form of the X and Y axis. An array is a set of values with distinct coordinates.
The X axis is usually referred to as 'width', the Y as 'height' and the Z as 'depth'.

Each coordinate in the array contains a value. For instance, in the array above, (1, 0) has the value "World".

**Arrays in Construct 2**

Arrays are an object, which means they are created like any other object by double clicking the canvas, or right clicking and selecting "Insert new object".

In Construct 2 (currently) arrays are 3 dimensional, although often they can be used as 2 dimensional or 1 dimensional. This means they have a X (width), Y (height) and Z (depth) axis.

**Size**

The size of an array determines how many values it can hold in each axis. Remember when I said Construct 2 was zero-based? This is one slightly confusing thing about arrays. **Size is not zero-based, but coordinates are.**

Let me give you an example. An array with a width, height and depth of 1 will only have 1 location where a value can be stored. This value will be at x = 0, y = 0 and z = 0.

Depth, height and width can be changed by clicking on the array object and altering their values in the Properties panel.
You can call the size of an array with an expression:

```
For Each Value
```

This condition is a combination of what we've learned. This loop cycles through every value in the array and will perform the actions specified for each one. There's a few expressions necessary for this to work:

- **Current X** - This will return the X coordinate of the value the loop is currently at.
- **Current Y** - See above
- **Current Z** - See above
- **CurrentValue** - this will return the value at (CurrentX, CurrentY, CurrentZ)

Using these expressions in conditions and actions can let you do some pretty cool stuff.
Physics

Created by Ashley


Construct 2 includes the Physics behavior, powered by Box2DWeb. This allows you to have objects moving with real-world physics. Physics can make your games really fun and engaging! Here's an overview of how you can get Physics to work in your game.

If you ever took a Physics class in school, you'll find some of the things you learned applicable to Construct 2's Physics. I'll still explain the basics in brief in case you haven't learnt the concepts before. Interested in some of the theory? You might want to read Wikipedia's article on Newton's laws of motion.

Lots of physics examples come with Construct 2! Click Browse all examples on the start screen, and all the physics demos filenames start with "Physics - ", e.g. "Physics - basics.capx". They're well worth having a look. The descriptions here will probably make a lot more sense if you've seen it in action first.

How to add Physics

Select an object you want to add Physics to. In the Properties Bar, click Add / Edit under Behaviors. Click the green plus icon and from the dialog pick Physics. All done!

We'll call any object with the Physics behavior added a "physics object".

Gravity

By default, gravity is present on physics objects, which accelerates all objects downwards. The default gravity is 10 (remember, the Y axis increases downwards in Construct 2). If you want to turn off gravity, you can use the Set gravity action on any physics object. Note: gravity applies to the whole "world". If you set gravity to 0 on one object, gravity is turned off for all objects.

Making the scenery

You don't want your game's floor to fall away off the bottom of the screen with gravity too! Even in "zero gravity" games, a piece of scenery will be slightly pushed back if something collides with it, and possibly start to rotate. (Newton's third law of motion: "every action has an equal and opposite reaction", which means the scenery is pushed back a bit too.)

Most of the time in our games we want to simulate the scenery being rock solid: not falling away with gravity, and not being pushed a little bit back by things hitting it. Set the physics object's Immovable property to Yes to simulate this.
The object is then simulated as if it has infinite mass. It won't go anywhere!

**Note:** Physics objects only interact with other physics objects. Adding the Solid behavior to the floor has no effect on physics objects. They ignore anything which does not also have the physics behavior. Physics objects will pass right through 'Solid' objects, unless you give them the Physics behavior and set them to immovable.

**The other physics properties**

Let's briefly go over the other properties in the physics behavior:

**Collision mask**

This sets the collision shape of the object. By default, it's use collision polygon. If you open up the image editor and click the collision polygon tool, you can change the bits of it that will collide. Be careful not to use too many points, or it could slow down the game!

If set to either bounding box or circle, the collision polygon from the image editor is ignored. Instead it'll either use a rectangle around the object (bounding box), or a circle which is useful for rolling objects like balls.

**Prevent rotation**

If enabled, the object will never rotate even if struck at a glancing angle. This might be useful if you want to control the angle of the player yourself. For example, in a platform game, you probably don't want your player to trip over and fall on their face every time they try to run.

**Density**

Density is used to determine the object’s mass. Mass defines how hard the object is to move. (Note that "weight" isn't the exactly correct term - weight is dependent on gravity, and objects still have mass in zero gravity. Objects with a large mass are still harder to move in zero gravity.) Your object’s mass is determined by its density multiplied by the area of its collision shape. So a really huge object has a much bigger mass than a small object, even if their densities are the same.

If you have a concrete block, you’d want to give it a much higher density than a block of foam!

**Friction**

Friction affects how much the object is slowed down by sliding against another object. No friction is like skating on ice, and high friction is like dragging a brick along concrete. It's harder work to pull
the brick along on concrete than on ice.

**Elasticity**
The elasticity (or "bounciness" or "restitution") affects how bouncy the object is. An object with high elasticity will bounce high when dropped on to the floor, and an object with no elasticity won’t bounce at all.

**Linear damping**
Objects move at the same speed in the same direction forever, unless something else affects them. Think of throwing a tennis ball in space - off it goes forever. (Newton’s first law of motion: "every object in a state of uniform motion tends to remain in that state of motion unless an external force is applied to it"). On Earth, forces like gravity, friction and air resistance tend to make this less noticeable.

In your physics game, you might find you push an object and it carries on forever, in accordance with Newton's third law. You might want to simulate some friction against the floor, or air resistance. Increasing the linear damping makes objects slow down gradually by themselves, ultimately to a stop. Zero linear damping is like in space - objects will carry on forever.

**Angular damping**
Angular damping is a very similar concept to linear damping, but instead to do with the object’s spin. Again, in space, a spinning object will spin at the same speed forever. Increasing the angular damping will make a spinning object gradually slow down until it is no longer spinning. Note the damping occurs regardless of how fast the object is moving.

**General physics tips**

**Performance**

Physics simulations are very CPU intensive. It can take a lot of processing to work out the proper motion. To make sure your game runs fast, it's recommended that you don't use too many objects at once. Over 100 physics objects moving at once is likely to slow your game down. Also, phones and tablets have much more limited processing power than a desktop computer. If you're targeting mobiles, you should be very conservative, and try not to have more than 20-30 physics objects.

Note that objects which have come *completely* to a stop, and are not moving or rotating *at all*, are "put to sleep" by the simulation. Then, they don't need processing any more. If the object is hit by another one it "wakes up" and starts using processing again. However, if the object is even moving in the slightest, it won't be put to sleep. For example, all the blocks in a teetering tower will remain awake. It's just something useful to bear in mind for performance: if you have hundreds of objects asleep and at most only ever 20-30 moving (even slightly), the game should still run well.

**Stability**

Physics simulations are not totally robust. If you simulate unrealistic things, like a gigantic concrete object hitting a block of foam at the speed of sound, the result is likely not to be realistic either. In fact, anything involving extreme forces is likely to cause the simulation to become unstable (unrealistic, e.g. objects moving through or inside each other).
Things like incredibly heavy objects piled on a stack of really light boxes, huge piles of objects, or very fast moving heavy objects tend to cause instability. Try to keep everything in your game at reasonable proportions.

The same also applies to object sizes. Very small (under 5 pixels) or very large (over 500 pixels) objects may not simulate realistically either. Try to keep all widths and heights in the 5-500 pixel range, and still avoid extreme proportions (e.g. a 5x500 sized object).

In other words, Physics will work best with objects of about the same size and mass interacting at relatively low speeds.

**Manual movement & other behaviors**

If you move objects by events (e.g. set X, set Y) or other behaviors (e.g. also adding 8 direction to a physics object), the physics simulation will do its best to keep up with what you've done. However, it is usually more realistic to achieve the same thing by applying forces and impulses to physics objects. This keeps everything "in the physics world" and realistic.

For example, if you use Set Position to move an object to the other side of the layout instantly, it has effectively teleported. That's not a realistic physical phenomenon, so the result may be unrealistic as well. To keep up, the physics behavior will spot this, and simulate the object suddenly moving extremely fast towards its destination point for one tick (about 1/60th of a second). That's an incredible amount of acceleration, speed, then deceleration. Remember, to keep a simulation stable, you should avoid extremes.

Although you can add the Platform behavior to a Physics object, the two tend not to get along very well. Again, it's better to achieve the same thing by applying forces. See the *Physics - rolling platformer* example provided with Construct 2 for a demonstration of how you can achieve this.

**Conclusion**

Physics can be really fun to have in your games, but don't forget: avoid having too many objects, avoid extremes, try to move things by forces and impulses alone - and it doesn't hurt to know some of the theory!
More Physics: Forces, impulses, torque and joints

Created by Ashley


The main fun of using Construct 2’s Physics behavior is in manipulating objects. You can accelerate, hit or constrain objects using forces, impulses and joints. Here’s how they work.

What are forces, impulses and joints?

A **force** is "any influence that causes an object to undergo a change in speed or direction" (according to Wikipedia). Usually you apply a force to an object over a long period of time to cause it to accelerate in a direction. For example, gravity is a constant downwards force.

An **impulse** is like a sudden impact on an object. If a ball is hit with a bat in your game, you’d apply an impulse rather than a force, for a sudden one-off strike. The important distinction is that forces are usually applied over time, whereas an impulse is a one-off hit. Forces make objects gradually accelerate, and impulses send them speeding off right away.

**Torque** is a rotational force. Instead of accelerating the object towards something, torque gradually speeds up the rate of rotation.

A **joint** constrains the motion between two objects. In other words, it connects two objects in some way. For example, you can "hinge" two objects together. The objects are still free to rotate independently of each other - but their position is constrained, because they are joined at a point.

How to use forces, impulses, torque and joints

The Physics behavior adds some actions to the object it was added to. You can find these in the event sheet editor mingled in with the object’s ordinary actions like Set Position.
Normally you’d apply **forces** and **torque** in a continuously true condition like *is left mouse button down, impulses* in triggers like *on collision with an object*, and create **joints** on *start of layout* or after creating a new object.

### Forces

When applying a force, you can specify the strength and direction of the force, as well as the point on the object it applies.

There are three actions to apply a force. They all essentially do the same thing, but some do a bit of math behind the scenes to make it easier to specify the force you want. **Apply force** can apply a force on the X and Y axes separately. **Apply force at angle** applies a force of a particular strength at an angle (and works out the X and Y components for you). **Apply force towards position** applies a force of a particular strength towards a position (and works out the angle, and then the X and Y components for you).

Here’s an image to demonstrate a force on The Pirate Princess:
The force (the blue arrow) can be specified in three ways: the X and Y components, the angle and strength (length of the arrow), or the strength and a position somewhere the arrow is pointing to. All three specify the same force, but each is convenient in different situations.

**Point of application**

You can also supply an image point when applying a force. This allows you to apply the force from a different position on the object. By default, it's applied from the object's origin, normally in the middle of the object. Notice the blue spot where the arrow starts in the image above is in the middle of The Pirate Princess.

Suppose we want to pull The Pirate Princess by his/her hook instead of from the origin. In the image editor, an image point can be placed on the hook. Then, we can provide that image point's name or number in the force-applying action. The force is then pulled from that point. See the image below: now the force is applied from the hook.

When you apply a force at the edge of an object, it's more likely to rotate, and move with that point in front. Forces from the middle tend to be more “floaty”, as if they're orbiting.

**Impulses**

The Apply Impulse actions are very similar in that you can apply an impulse with X and Y components, towards an angle, or towards a position. You can also specify the impulse point from an image point on the object. Again, applying impulses from the edge of the object are more likely to send the object off spinning.

Remember impulses are one-off hits - you probably only want to use them in triggered events like *on collision with an object.*
Torque

Torque is a rotational force. Instead of giving an X and Y component, you simply enter a single value. A positive torque accelerates the object in a clockwise direction, and a negative torque accelerates the object in an anticlockwise direction.

Torque affects the object's angular velocity. That's how many degrees per second it is rotating. Again, a positive value is clockwise rotation, and negative is anticlockwise.

You can apply torque towards an angle. This applies the rotation force either clockwise or anticlockwise towards the given angle - whichever's shortest. You can also apply torque towards a position. While that might sound like an odd thing to do, it simply calculates the angle to that position, then applies the torque towards that angle.

You can't apply torque from a position. Rotation affects the whole object!

Joints

There are two kinds of joints available to the Physics behavior.

Revolute joint

A revolute joint is like a hinge: an object is pinned on to another object. The objects are free to rotate, but are joined together at that point. See the Physics - revolute joint example provided with Construct 2 for a demonstration. Revolute joints are created with the Create revolute joint action.

You only need to create the joint once, either on start of layout or when the objects are created or need to be attached, then it applies to the objects forever. You can join the objects from an image point, so the other object doesn't have to be attached to the origin. You don't need to apply an image point for the other object - it's attached from its current position, so make sure it's in the right place before the Create revolute joint action.

Distance joint

A distance joint makes two objects act as if they are joined by a pole. They always stay exactly the same distance apart, although they can rotate about their attachment points. See the Physics - distance joint example provided with Construct 2 for a demonstration. Distance joints are created with the Create distance joint action.
Note the connecting pole is not shown by the action itself! If you use this action on its own, there is nothing visible joining the objects, although they act as if there was a pole joining them. The above picture and the distance joint demo use a sprite to display the pole, so it's easier to see how the joint works. You might want to do something similar in your games, since it can look odd having objects joined by an invisible pole.

You can specify image points for both objects being attached. In the action's parameters, This image point refers to a point on the object the physics behavior is added to, and That image point refers to an image point on the other object. This allows you to connect two objects by their edges, for example.

You can also specify the damping ratio and spring frequency. The spring frequency makes the connecting pole act more like a large spring. If you set it to 4 Hz and one of the objects is struck, the connection between the two objects will "bounce" a little at about 4 oscillations a second. The rate the oscillations diminish is set by the damping ratio. A damping ratio of 0 means the spring will keep oscillating forever, and 1 means the spring will stop oscillating almost immediately.

Don't forget...

Newton's second law says "The relationship between an object's mass m, its acceleration a, and the applied force F is F = ma". In other words, if you apply a force of 100 to a small object with a low mass, you get a lot more acceleration (a) than if you applied the same force to a big object with a large mass. It should be obvious that big, heavy objects are harder to move and tend to move slower! Therefore, you may find yourself needing to use smaller forces for small objects, and larger forces for big objects. The same is true of impulses and torque.

Conclusion

Forces, impulses and torque allow you to throw objects around in your game, setting them accelerating, rotating and flying. Joints allow you to connect objects up in fun ways. You can even make strange contraptions! Check out the Physics - vehicle example for one idea. You could try connecting a whole series of objects up by distance joints and see what happens. The possibilities are endless - and it's great fun to play with, too!
Delta-time and framerate independence

Created by Ashley


Framerate independent games are games that run at the same speed, no matter the framerate. For example, a game might run at 30 FPS (Frames Per Second) on a slow computer, and 60 FPS on a fast one. A framerate independent game progresses at the same speed on both computers (objects appear to move at the same speed). On the other hand, a framerate dependent game progresses at half the speed on the slow computer, in a sort of slow-motion effect. Making framerate independent games is important to make sure your game is enjoyable and playable for everyone, no matter what kind of computer they have. Games which slow down when the framerate dips can severely affect gameplay, making players get frustrated and quit!

This tutorial describes how you can make your game framerate independent. The same technique also enables time scaling, which allows for deliberate slow-motion effects and easy pausing.

The \( dt \) system expression

The key to framerate independence is the \( dt \) system expression. \( dt \) stands for delta-time. Delta means a change in a quantity, so delta-time means the change in time. It is the time, in seconds, since the last tick.

For example, at 100 fps \( dt \) will be 0.01 (one hundredth of a second), and at 10 fps \( dt \) will be 0.1 (one tenth of a second). In practice, \( dt \) varies tick-by-tick, so it is unlikely to be exactly the same value for long periods of time.

Notice that if you add \( dt \) to a variable every tick, it adds 1 every second, because the time between all the ticks over a period of 1 second must add up to 1! Here's an example showing just that. (Adding \( dt \) to an object's instance variable is also a handy way to have a timer in an object.)

How to use \( dt \)

Typically, framerate dependent motion is done with an event like this:

```
1  System Every tick piggy Set X to Self.X + 1
```

Every tick (once per frame), the object moves one pixel to the right. Notice that at 30 FPS this means 30 pixels per second, and at 60 FPS this means 60 pixels per second. Those are different speeds, depending on the framerate.

Remember from the example above that \( dt \) always adds up to 1 every second. So if we change the event to the following:

```
1  System Every tick piggy Set X to Self.X + 60 * dt
```

...
...the object will move to the right at 60 pixels per second at any framerate. Since \( dt \) adds up to 1 every second, \( 60 \times dt \) adds up to 60 every second. This means at both 30 FPS and 60 FPS our object moves to the right 60 pixels every second - the same speed, regardless of framerate.

**Use \( dt \) everywhere**

Any time you move an object at a steady speed, you need to use \( dt \) in that way to achieve framerate independence. For example, Sprite's *Move forward* action takes a number of pixels to move the object forward. If you constantly move the object forwards, you could move it forward \( 60 \times dt \) pixels to move it at 60 pixels per second at its current angle.

**Behaviors already use \( dt \)**

All of Construct 2's behaviors use \( dt \) in their internal movement calculations. That means anything moved by behaviors like Platform and 8 Direction don't need any special treatment - they do this already!

Note that *Physics* is an exception. By default it does *not* use \( dt \), and therefore is framerate dependent. This is because \( dt \) usually has small random variations. These variations can make the same setup in a physics game give different results even if exactly the same thing is done twice. This is often annoying for physics games, so by default it is framerate dependent. However, you can enable use of \( dt \) by using the *Set Stepping Mode* physics action on start of layout, and choose *Framerate independent*.

**Timescaling**

A really cool feature in Construct 2 is **timescaling**. This allows you to change the rate time passes at in the game, also known as the *time scale*. You can set the time scale with the system *Set Time Scale* action. A time scale of 1 means normal speed. 0.5 means half as fast, and 2.0 means twice as fast. If you set your game's time scale to 0.1, it's going ten times slower but still smoothly - a nice slow-motion effect!

Timescaling works by changing the value returned by \( dt \). This means behaviors are affected, and any movement using \( dt \). If you don't use \( dt \) in your movement calculations (like the first event above) the motion is not affected by the time scale! So to use time scaling, you simply have to use \( dt \) properly in all movement.

**Pausing**

You can set the time scale to 0. This stops all motion. It's an easy way to pause the game. Set it back to 1 and the game will resume.

It's also a good way to test you have used \( dt \) correctly. If you have used it correctly, setting the time scale to 0 will stop *everything* in the game. If you have not used it correctly, some objects might continue to move, even though the game is supposed to be paused! In that case you can check how those objects are moved, and make sure you are using \( dt \) properly.
Other kinds of movement

It's important to realise that $dt$ must be used for all kinds of motion. This includes rotation and acceleration.

Rotation

Similar to before, the following event rotates the piggy 1 degree every tick:

```
1  System  Every tick  piggy  Rotate 1 degrees clockwise
```

This is 30 degrees per second at 30 FPS, and 60 degrees per second at 60 FPS - again, different speeds for different framerates. Using $dt$ in the same way solves the problem again. This way the piggy rotates at 60 degrees per second regardless of the framerate:

```
1  System  Every tick  piggy  Rotate 60 * $dt$ degrees clockwise
```

Acceleration

Acceleration is also fairly straightforward. Usually this only applies when you are making a custom movement via events.

If you have a speed variable, your object will be moving at $Object.Speed \times dt$ pixels per tick. Your object's speed variable therefore contains a speed in pixels per second.

Suppose you want to accelerate the object by 100 pixels per second over one second. You simply need to add $100 \times dt$ to the object's speed variable every tick, and it will accelerate in a framerate independent way. In other words, you use $dt$ both to adjust the object's position, and to adjust the object's speed.

Advanced considerations

Minimum framerate

At very low framerates, $dt$ can become very large. For example, at 5 FPS, $dt$ is 0.2. An object moving at 500 pixels per second is therefore moving 100 pixels per tick. This can cause it to "teleport" through walls or miss collisions with other objects.

Games are usually unplayable at such low framerates, but it is even worse if they become unstable like that. To help the game stay reliable even at very low framerates, Construct 2 does not let $dt$ get larger than 0.1. In other words, below 10 FPS, $dt$ stays at 0.1. This does also mean below 10 FPS the game starts going in to a slow-motion effect (described earlier as one of the issues of framerate dependent games), however this is usually a better result than the "teleporting objects" problem.

Random variation

As mentioned with physics, $dt$ usually has small and random variations, usually due to imperfect timers in the computer. Like with physics, this can also cause slight random variations in your game.
However, usually the effect is negligible and much less noticeable than with physics. It's recommended that you always use \textit{dt} in your games unless exact precision is required (which is rare).

**Object time scales**

You can set a time scale for an individual object, via the system \textit{Set object time scale} action. This allows you to have, for example, a game running in slow-motion with a time scale of 0.3, but still have the player acting at full speed with a time scale of 1. This is achieved by setting the time scale to 0.3, then using \textit{Set object time scale} to set the player's time scale. The system expression \textit{dt} is only affected by the game time scale. Objects have their own \textit{dt} expression (e.g. \texttt{Player.dt}) which you must use instead of the system \textit{dt} for all motion relating to that object. This way, there are now two values of \textit{dt}: one for the game, and one for the player. Since they return different values, different parts of the game can progress at different rates.

In this example, to return the player back to game time, use the system \textit{Restore object time scale} action.

**Conclusion**

It's important to design your game from the start using \textit{dt}. This improves the gameplay, ensuring the pace never slows down in particularly intense parts of the game. As an extra bonus, you can also use the \textbf{time scaling} feature, easily \textbf{pause} the game, and even control object's individual time scales.

Don't forget behaviors already use \textit{dt} (except with Physics where you must turn it on yourself). If you use behaviors to control all motion in your game, you don't have to worry about \textit{dt} at all! However, most games have some event-controlled motion, and it's important to remember how to make it framerate independent.
Upload your game to Dropbox

Created by Ashley

http://www.scirra.com/tutorials/42/upload-your-game-to-dropbox

Have you finished your game and want to share it with the world? If you have your own website and some HTML know-how, you probably are comfortable integrating the exported files to your site. However, if you don't have your own website, you can share your games on Dropbox!

**Get a Dropbox account**

If you don't have one already, head to Dropbox.com and sign up an account. It's free, and you get about 2GB of storage (that's plenty for our games!). You can download the desktop app if you like, but you don't need it for this. We're going to upload a game using the online uploader instead.

**Export your project**

Open the game you want to share. In the File menu, click **Export project**.

The default settings should be fine, so click OK. Construct 2 fills a folder with all the necessary files. If Construct 2 asks you, click to open the folder. Otherwise, browse to the folder Construct 2 exported to, and open it. By default, it exports to a new folder on your desktop. You should see some exported files a bit like this:
We need to upload all these files for the game to work.

**Log in to Dropbox**

Log in to [Dropbox.com](https://www.dropbox.com) with your new (or existing) Dropbox account.

By default all your files on Dropbox are private. To share them with the world, we need to put them in the **public** folder. In the **Files** tab, click the **public** folder to open it.

Click **New folder**. We might want to upload another game later, so let's keep one game to a folder. Name the folder after your game, and click **Create**. Open this folder.

**Uploading the game files**

Now you're in your game's folder, click **Upload**. Click **Choose files**, and select everything from your exported folder. Upload those.

Dropbox won't let you upload an entire folder at once, so we need to separately create the **images** folder. Create a new folder inside this folder, call it **images**, and open it.

Click **Upload** again, and this time select the entire contents of the exported **images** folder and upload them.
### Getting a public link

Go back to the game's folder where `index.html` is. Hover the mouse over `index.html`, and click the little drop-down arrow to the right. Select **Copy public link**. It'll give you a link - copy it to the clipboard.

### Sharing the link

You're now armed with a public link to your game that you can paste anywhere. Paste it to your friends, send it by email, link to it from your blog, tweet it, put it on Facebook, Google+ it, do what you like! Your game is now accessible to the whole world. Have fun! :)

### Using the desktop application

If you like, you can also install the desktop application for Dropbox from their website. This gives you a folder on your computer which is synced with your Dropbox account. You can then copy and paste the entire exported folder to the Dropbox public folder on disk, wait for it to upload, then right-click `index.html` and copy a public link - all from Windows. You might find this a bit quicker, if you're happy to have Dropbox running on your computer.
Q: How do I move a sprite in a sine wave pattern?
   A: Use $\sin(time \times modifier) \times modifier$. Play with the modifiers to adjust values like speed and distance.

Q: How do I use a single image tile sheet?
   A: Try right click and ‘import sprite strip’ in the animation frames bar - by default, the one at the bottom, showing the animation frames.

Q: How do I loop an animation?
   A: You have to check "Loop" in the animation properties (select the name of your anim in the animation window, check the property tab on the left of the screen).