

gameBlender Documentation

Last modified 01/03/22 16:24:49 cw

Carsten "Calli" Wartmann

Ton "Evil" Roosendaal

Willem "Zycho" Zwarthoed

V0.99b ;-) for Blender 2.12 Edition

Copyright © 2000, 2001 by Not a Number BV, Amsterdam

gameBlender Documentation: Last modified 01/03/22 16:24:49 cw

by Carsten "Calli" Wartmann, Ton "Evil" Roosendaal, and Willem "Zycho" Zwarthoed

V0.99b ;-) for Blender 2.12 Edition

Copyright © 2000, 2001 by Not a Number BV, Amsterdam

Table of Contents

1. Introduction.....	9
1.1. Intention of this document	9
1.2. Why gameBlender, what are the strengths ?	9
1.3. Simple gameLogic Tutorial.....	9
2. The gameEngine.....	17
2.1. Blenders gameEngine "Ketsji"	17
2.2. Options for the gameEngine.....	17
2.3. Command line options for the gameEngine.....	18
2.4. The RealtimeButtons.....	20
2.5. Properties	22
2.6. Settings in the MaterialButtons	23
2.6.1. Specularity settings for the gameEngine	24
2.7. Lamps in the gameEngine.....	25
2.8. The blender laws of physics.....	26
2.9. Expressions	27
2.10. SoundButtons.....	29
2.11. SoundWindow	30
2.12. Performance and design style issues.....	31
3. Game LogicBricks	33
3.1. Sensors	33
3.1.1. Always Sensor	33
3.1.2. Keyboard Sensor.....	34
3.1.3. Mouse Sensor	36
3.1.4. Touch Sensor	36
3.1.5. Collision Sensor.....	37
3.1.6. Near Sensor	38
3.1.7. Radar Sensor.....	39
3.1.8. Property Sensor	40
3.1.9. Random Sensor.....	42
3.1.10. Ray Sensor.....	42
3.2. Controllers.....	43
3.2.1. AND Controller	44
3.2.2. OR Controller	44
3.2.3. Expression Controller.....	44
3.2.4. Python Controller	45
3.3. Actuators	46
3.3.1. Motion Actuator	46
3.3.2. Constraint Actuator	48
3.3.3. IPO Actuator.....	50
3.3.4. Camera Actuator.....	51
3.3.5. Sound Actuator.....	52

3.3.6. Property Actuator	53
3.3.7. Edit Object Actuator.....	54
3.3.8. Scene Actuator.....	57
3.3.9. Random Actuator.....	58
4. UV Texturing.....	63
4.1. Available file formats	63
4.2. Handling of resources	63
4.3. The UV Editor.....	65
4.3.1. The ImageWindow	65
4.3.2. The Paint/FaceButtons.....	67
4.3.3. Bitmap text in the game engine	69
5. Python.....	71
5.1. The TextWindow	71
5.2. Python for games.....	72
5.2.1. Basic gamePython	73
5.3. Examples for gamePython	74
5.3.1. Simple visibility check	74
5.3.2. Mouse cursor with python	75
5.4. gamePython Documentation per module	76
5.4.1. GameLogic Module.....	77
5.4.2. Rasterizer Module	77
5.4.3. GameKeys Module	78
5.5. Standard methods for LogicBricks	78
5.5.1. Standard methods for Sensors	79
5.5.2. Standard methods for Controllers.....	80
5.5.3. Standard methods for gameObjects.....	80
Index.....	83

List of Tables

2-1. Valid expressions	-999
2-2. Arithmetic expressions	28
2-3. Boolean operations	28
2-4. Expression examples	29

List of Figures

1-1. Our Player	10
1-2. The loaded start scene	10
1-3. Object attributes to set	11
1-4. LogicBricks to move our player forward	12
1-5. Changing the LogicBrick type	12
1-6. LogicBricks to steer the player	14
1-7. Jump!	14
2-1. The GameMenu	17
2-2. Blender command line options	18
2-3. RealtimeButtons left part	20
2-4. Example of some LogicBricks	22
2-5. Defining properties	22
2-6. Material settings for dynamic objects	24
2-7. Specularity settings	25
2-8. LampButtons, settings for gameBlender	25
2-9. The SoundButtons	29
2-10. 3D sound demo	30
2-11. The SoundWindow	31
3-1. Common elements for Sensors	33
3-2. Pulse Mode Buttons	34
3-4. Schematic of pulses for a near sensor	38
4-1. The ToolsMenu	64
4-3. The Image Window	65
4-4. The Paint/FaceButtons	67
4-5. The special menu for the FaceSelectMode	67
5-1. The TextWindow	71
5-2. LogicBricks for a first gamePython script	73
5-3. First Script	73
5-4. Visibility Script	74
5-5. LogicBricks for the visibility script	75
5-6. Mouse cursor script	75
5-7. Logic bricks for the mouse script	76

Chapter 1. Introduction

1.1. Intention of this document



This document is in its current state a draft and needs input from the developers, content and the users. Mail Carsten@blender.nl (<mailto:carsten@blender.nl>) for suggestions, error reports, critics, etc.

Please don't complain when you print the PDF and next week we update the documentation. Save some trees and read it on screen until it is (more) finished.

This document is intended to provide reference documentation to the 3D-realtime part of Blender, known and referred in this document as gameBlender.

It does not contain bigger tutorials, that will be part of coming documentation. It includes small demo files and examples, to show the functions in a working Blendfile.

This document is available in *PDF (Portable Document Format, *.pdf)* (print/gBlenderDoc.pdf.gz) and as well structured HTML with TOC and index at <http://www.blender.nl/gameBlenderDoc/>



Warning 2

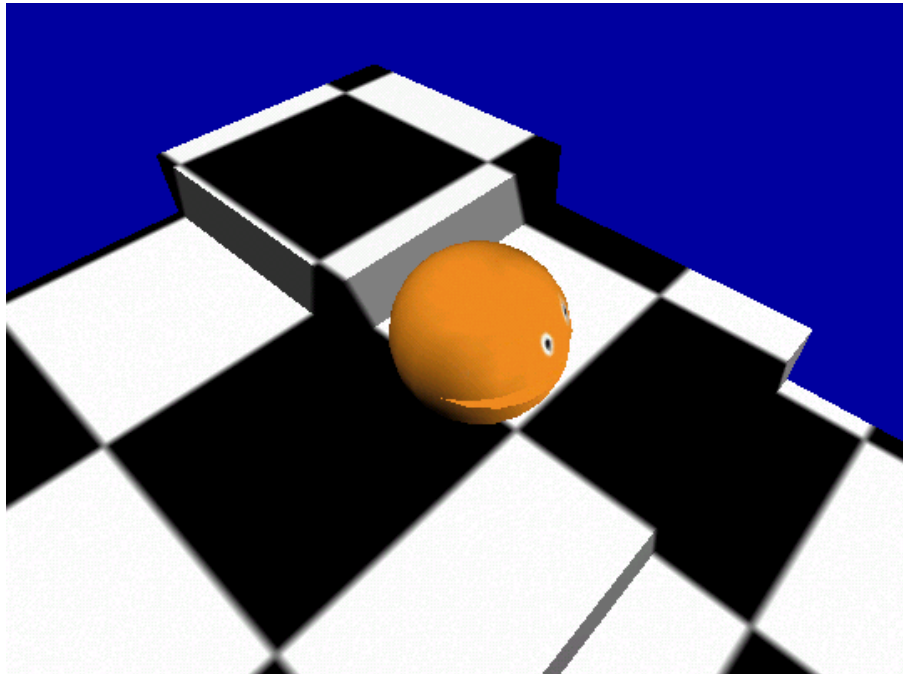
Please take care with the Python parts, they are not 100% ready or tested (in terms of documentation!). So if you have problems don't hesitate to mail me! Carsten@blender.nl (<mailto:carsten@blender.nl>)

1.2. Why gameBlender, what are the strengths ?

- Integrated environment, with modeling, animation and game player.
- Build-in Physics (rigid body dynamics) and Collision simulation
- Easy Interactivity with predefined Sensors and other Logic Bricks
- Powerful scripting language Python for more advanced game play control
- True Multiplatform, All flavors of Windows, Linux, FreeBSD, BeOS, Irix and more

1.3. Simple gameLogic Tutorial

Figure 1-1. Our Player



In this small tutorial you will get an overview of:

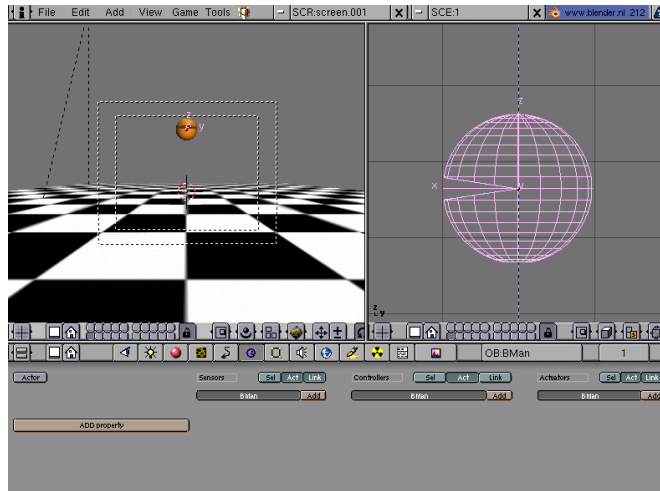
- To set object attributes
- To add LogicBricks
- To connect LogicBricks
- Basic use of the Keyboard, Always and Touch Sensor

You will not learn here how to use Blender as modelling or animation tool. For this information please refer to our online help (<http://www.blender.nl/help/beginners.php>) and our printed documentation (<http://www.blender.nl/shop/>).

Preparings

Start Blender and load the prepared start scene B-Man1.blend (blends/simple/B-Man1.blend). You can use the F1KEY or the filemenu to load a scene. After loading you should see a scene like in [Figure 1-2](#).

Figure 1-2. The loaded start scene



The left 3DWindow is a textured view from the camera. The right window is a wire frame side view of the player character. It has the axis made visible and is selected, indicated by the pink color of the wire frames.

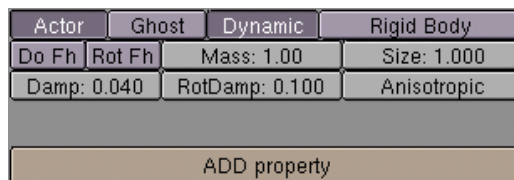
The broad lower window are the RealtimeButtons, you can call then with F8. Here most of the work for interactive realtime 3D graphics in Blender will take place.

Now we can start the gameEngine moving the mouse cursor to the textured 3DWindow and press PKEY. Because we have not defined any interactivity you will only see the helper lines dissappear and the sky becomming blue instead of grey. Press ESC to stop the gameEngine again.

Make the actor falling

Now click with the left mouse button on the "Actor" button in the RealtimeButtons. Also activate the appearing "Dynamic" button. This defines the selected object as a actor which is handled by the build in physics of the gameEngine. If you accidentally deselected the player (black wire frame) please reselect it with the right mouse button or re-load the scene.

Figure 1-3. Object attributes to set



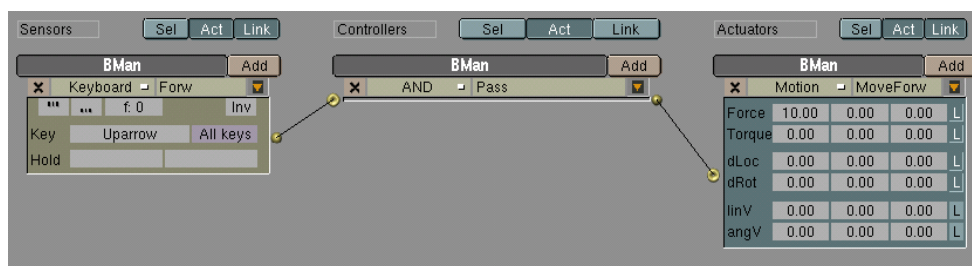
Now we can run the gameEngine again. Press PKEY with the mouse over the textured view and you will see that the player falls to the floor and bounces a few times until he gets to rest. Press ESC to stop the gameEngine.

Make it move

The RealtimeButtons (F8) are logically divided into four columns. The leftmost we already used to set the object parameters to make it fall. The three right columns are used for building the interactivity into our game.

Now lets move the player on our request.

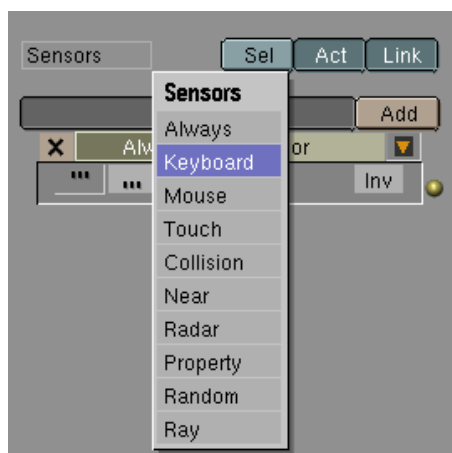
Figure 1-4. LogicBricks to move our player forward



The three parts are labeled as "Sensors", "Controllers" and "Actuators". You can think of Sensors as the senses of a life form, the Controllers are the brain and the Actuators are the muscles.

Now press the "Add" button for each row one time with the left mouse button to make one LogicBrick for the Sensors, Controllers and Actuators.

Figure 1-5. Changing the LogicBrick type



The types of the added LogicBricks are nearly correct, for our first task, only the first one needs a change. Press and hold the MenuButton now labeled with "Always" and choose "Keyboard" from the pop-up menu (see [Figure 1-5](#)).

Now click with the left mouse into the "Key" field. The text "Press any key" appears. Press the key you want to use to move the player forward (I suggest UPARROW).

Now have a closer look at the Motion Controller. We will now define how the player should move. The first line of numbers labeled "Force" defines how much force will be applied when the Motion Controller is active. The three numbers stand for the forces in X, Y, and Z-Axis.

If you look at the wire frame view of the player you see that the X-Axis is pointing forward on the player. To move forward we need to apply a positive force along the X-Axis. To do so click and hold on the first number in the "Force" row with the left mouse. Drag the mouse to the right to increment the value to 10.00. You can hold the CTRL key to snap the values to decadic values. Another way to enter an exact value is to hold SHIFT while clicking the field with the left mouse. This allows you to enter a value using the keyboard.

We have now nearly the configuration shown in [Figure 1-4](#). We now need to "wire" or connect the LogicBricks.

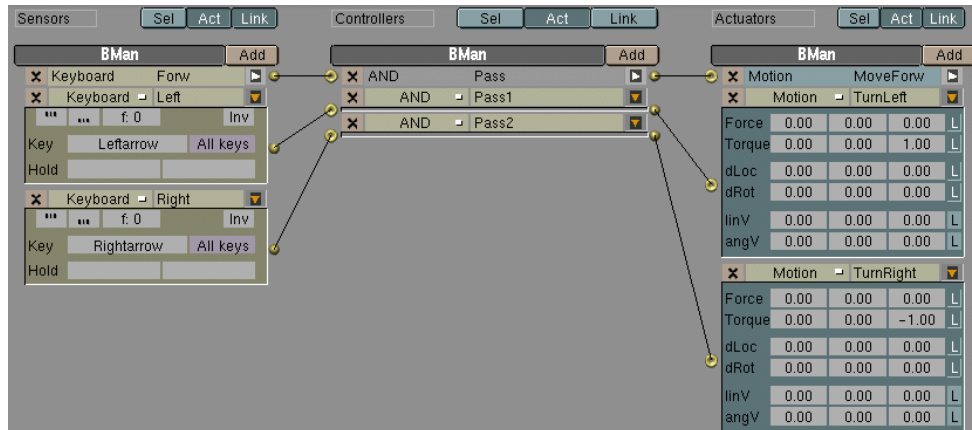
Click and hold with the left mouse button on the yellow ball attached on the Keyboard Sensor and drag the appearing line to the yellow ring on the AND Controller. Release the mouse and the LogicBricks are connected. Now connect the yellow ball on the right side of the AND Controller with the ring on the Motion Controller.

To delete a connection move the mouse over the connection. The line is now drawn highlighted and can be deleted with a XKEY or DEL keypress.

Now press PKEY to start the gameEngine and when you press the UPARROW key, the player moves towards the camera.

More control

Figure 1-6. LogicBricks to steer the player

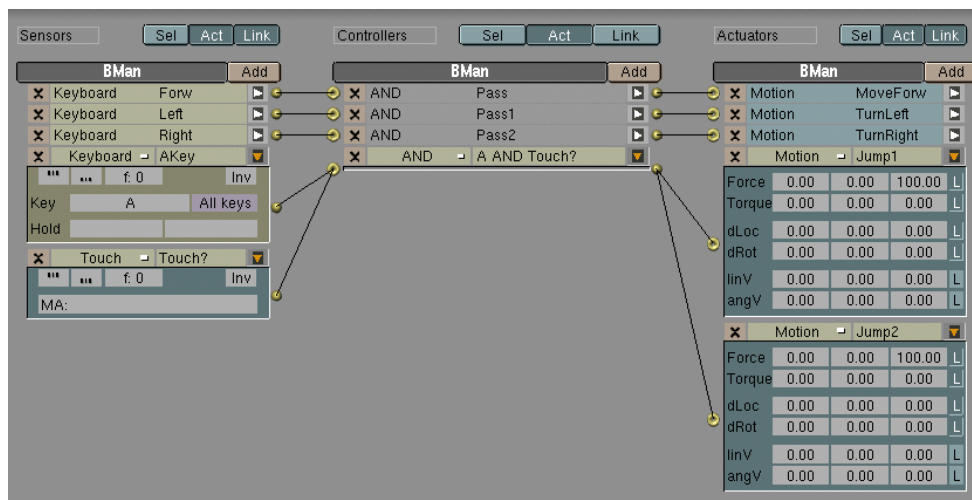


Now add more LogicBricks like shown in [Figure 1-6](#). These LogicBricks will allow you to steer your player with the Cursorkeys. Note that we use the "Torque" row from the Motion Actuator to turn the player.

Jump!

To add some more degrees of freedom and to show more than one sensor as input for a Controller, we will now make the player jump.

Figure 1-7. Jump!



Add the LogicBricks shown in [Figure 1-7](#) to the player. The jump is triggered by a Keyboard Sensor. But there is also a Touch Sensor connected to the same AND Controller. The Touch Controller only gives an impulse when the object *touches* something. This constellation now reads as: " AKEY is pressed AND the player touches the ground, THEN give an impulse to the Motion Actuator". This way we make sure that you can't jump while you are in the air. Try to delete the link from the Touch Sensor to the AND Controller and see what happens...

The Controller for the jumping is connected to two Motion Controllers, both have a force of 100.00 for the Z-Axis. Because 100.0 is the maximum for one Motion Actuator we simply use two Actuators to make the jump high enough.

Get the complete scene here: [B-Man6.blend](#) ([blends/simple/B-Man1.blend](#)).

Chapter 2. The gameEngine

2.1. Blenders gameEngine "Ketsji"

catch-y (kǎch'ě, kǎch'ě)

adj. **catch-i-er**, **catch-i-est**.

1. Attractive or appealing: *a catchy idea for a new television series.*
2. Easily remembered: *a song with a catchy tune.*
3. Tricky; deceptive: *a catchy question on an exam.*
4. Fitful or spasmodic: *catchy breathing.*

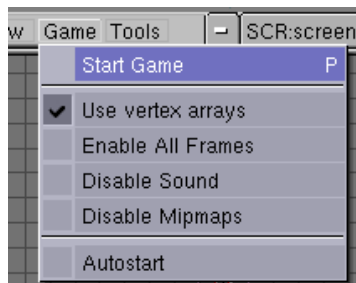
catch'i-ness *n.*

Ketsji is the name of the gameEngine. Technically the gameEngine is a framework with a collection of modules for interactive purposes like physics, graphics, logic, sound and networking. Functionally the gameEngine processes virtual reality, consisting of content (the world, it's buildings) and behaviors (like physics, animation and logic). Elements in this world - also called GameObjects - behave autonomously by having a set of tools called LogicBricks, and Properties. For comparison the Properties reflect memory, the Sensors are the senses, the Controllers are the brain and the Actuators allow for actions to the outside world (i.e. muscles).

At the moment the Controllers can be scripted using python, or simple expressions. The idea is that the creation of logical behavior can be edited in a more visual way in the future, so the set of controllers expands with AI state machines, etc. Controllers could be split in control centers, like an audio visual center, motion center, etc.

2.2. Options for the gameEngine

Figure 2-1. The GameMenu



With this menu you can change options during editing of your scene. Currently only the "Autostart" option is saved with the file.

Options from the GameMenu

Start Game (PKEY)

Start the gameEngine

Use vertex arrays

Enable (checked) or disable the use of vertexarrays. Vertexarrays normally speed up the calculation on complex scenes. If your OpenGL system don't support vertexarrays you can switch them off using this option

Enable All Frames

With this option checked the gameEngine runs on 50 hertz without dropping frames. This is usefull while record to a Targa-Sequence or when you need to make sure that all collisions are calculated without loss on slower computers

Disable Sound

Disable audio when checked

Disable Mipmaps

Don't use mipmap, this can speedup the game

Autostart

Enable Autostart on load

2.3. Command line options for the gameEngine

When Blender is called with the option "-h" on a command line (shell window or DOS window) it prints out the command line parameters.

Figure 2-2. Blender command line options

```
bash-2.00$ blender -h
Blender V 2.12
Usage: blender [options ...] [file]

Render options:
  -b <file>      Render <file> in background
  -S <name>      Set scene <name>
  -f <frame>     Render frame <frame> and save it
```

```
-s <frame> Set start to frame <frame> (use with -a)
-e <frame> Set end to frame (use with -a)<frame>
-a          Render animation
```

Animation options:

```
-a <file(s)> Playback <file(s)>
-m          Read from disk (Don't buffer)
```

Window options:

```
-w          Force opening with borders
-p <sx> <sy> <w> <h> Open with lower left corner at <sx>, <sy>
              and width and height <w>, <h>
```

Game Engine specific options:

```
-g fixedtime      Run on 50 hertz without dropping frames
-g vertexarrays  Use Vertex Arrays for rendering (usually faster)
-g noaudio       No audio in Game Engine
-g nomipmap     No Texture Mipmapping
-g linearmipmap  Linear Texture Mipmapping instead of Nearest (default)
```

Misc options:

```
-d          Turn debugging on
-noaudio    Disable audio on systems that support audio
-h          Print this help text
-y         Disable OnLoad scene scripts, use -Y to find out why its -y
```

bash-2.00\$

Command line options for gameBlender

-g fixedtime

With this option the gameEngine runs on 50 hertz without dropping frames. This is usefull while record to a Targa-Sequence or when you need to make sure that all collisions are calculated without loss on slower computers

-g vertexarrays

Disable the use of vertexarrays. Vertexarrays normally speed up the calculation on complex scenes. If your OpenGL system don't support vertexarrays you can switch them off using this option

-g noaudio

Disable audio

-g nomipmap

Don't use mipmap, this can speedup the game

-g linearmipmap

Linear Texture Mipmapping instead of Nearest (default)

2.4. The RealtimeButtons



The RealtimeButtons are meant for making interactive 3D worlds in Blender. Blender acts as a complete development tool for interactive worlds including a gameEngine to play the worlds. All is done without compiling the game or interactive world. Just press PKEY and it runs in realtime. The main view for working with gameBlender is the RealtimeButtons (). Here you define your LogicBricks, which add the behavior to your objects.

Figure 2-3. RealtimeButtons left part

Actor		Ghost	Dynamic	Rigid Body	
Do Fh	Rot Fh	Mass: 1.00		Size: 1.000	
Damp: 0.800		RotDamp: 0.400		Anisotropic	
x friction: 1.000		y friction: 1.000		z friction: 1.000	
ADD property					
Del	Float ▾	Name:prop	0.00	D	
Del	String ▾	Name:stringprop	I am a string!	D	
Del	Timer ▾	Name:time	160	D	

 The word "games" is here used for all kinds of interactive 3D-content; Blender is not limited to make and play games

The RealtimeButtons can be logically separated in two parts. The left part contains global settings for gameObjects.

This includes settings for general physics, like the damping or mass. Here you can also define if an object should be calculated with the build-in physics, as an actor or should be handled as object forming the level (props on a stage).

Settings for gameObjects

Actor

Activating "Actor" for an object causes the gameEngine to evaluate this object. The Actorbutton will spawn more buttons described below. Objects without "Actor" can form the level (like props on a stage) and are seen by other actors as well.

Ghost

Ghost objects that don't resitute to collisions, but still trigger a collision sensor.

Dynamic

With this option activated, the object follows the laws of physics. This option spawns new buttons that allow you to define the objects attributes in more detail.

Rigid Body

The "Rigid Body" button enables advanced physics by the gameEngine. This makes it possible to make spheres roll automatically when they make contact with other objects and the friction between the materials is non-zero. The rigid body dynamics are a range of future changes to the gameEngine.

Do Fh

This button activates the Fh mechanism (see [Section 2.6](#)). With this option you can create a floating or swimming behavior for actors.

Rot Fh

With this option set the object is rotated in such a way that the z-axis points away from the ground when using the Fh mechanism.

Mass

The mass of a dynamic actor has an effect on how the actor reacts when forces are applied to it. You need bigger force to move a heavier object. Note that heavier objects don't fall faster! It is the air drag that causes a difference in falling speed in our environment (without air, e.g. on the moon, a feather and a hammer fall at the same speed). Use the "Damp" value to simulate air drag.

Size

The size of the bounding sphere. The bounding sphere determines the area with which collisions can occur. In future versions this will not be limited to spheres anymore.

Damp

General (movement) damping for the object. Use this value for simulating the damping an object recieves from air or water. In a space scene you might want to use very low or zero damping, air needs a higher damping; use a very high damping to simulate water.

RotDamp

Same as "Damp" but for rotations of the object.

Anisotropic

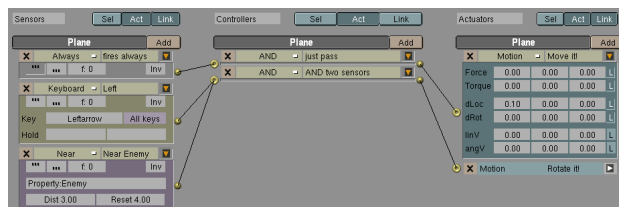
When an actor moves on a surface you can define a friction between the objects. Friction will slow down objects, because it is a force that works against any existing forces in the direction of the surface. It is controlled in the dynamic material settings (MaterialButtons F5, see [Section 2.6](#)). This friction works equally in all directions of movement.

With the "Anisotropic" option activated you can control the friction independently for the three axes. This is very helpful for racing games, where for example the car receives little friction in driving direction (because of the rolling tires) and high friction sliding to the side (Example file: blends/AnisotropicFriction.blend).

Below the object settings you define the Properties of a gameObject. These Properties can carry values, which describe attributes of the object like variables in a programming language. Use "ADD property" to add properties (see [Section 2.5](#)).

The right part of the RealtimeButtons is the command center for adding gamellogic to your objects and worlds. The logic consists of the Sensors, Controllers and Actuators.

Figure 2-4. Example of some LogicBricks



Sensors are like the senses of a life form; they react on keypresses, collisions, contact with materials (touch), timer events or values of properties.

The Controllers are collecting events from the sensors and are able to calculate them to a result. These are much like the mind or brain of a life form. Simple Controllers just do an AND. An example is to test if a key is pressed AND a certain time has passed. There are also OR Controllers and you also can use Python scripting and expressions in the Expression Controller to create more complex behavior.

The Actuator actually perform actions on objects. A Motion Actuator for example is like a muscle. This muscle can apply forces to objects to move or rotate them. There are also Actuators for playing predefined animations (via IPOs), which can be compared to a reflex.

The logic is connected (wired) with the mouse, Sensors to Controllers and Controllers to Actuators. After that you are immediately able to play the game! If you discover something in the game you don't like, just stop the gameEngine, edit your 3D world and restart. This way you can drastically cut down your development time!

2.5. Properties

Properties carry information bound to the object, similar to a local variable in programming languages. No other object can normally access these properties, but it is possible to copy Properties with the [Property Copy Actuator](#) (see [Section 3.3.6](#)).

Figure 2-5. Defining properties

ADD property					
Del	Bool ▾	Name:BoolProp	True	False	D
Del	Int ▾	Name:IntProp	0		D
Del	Float ▾	Name:FloatProp	0.00		D
Del	String ▾	Name:StringProp	I am		D
Del	Timer ▾	Name:TimeProp	0		D
Del	String ▾	Name:Result	Water.		D

The big "ADD property" button adds a new Property. By default a Property of the float type is added. Delete a Property with its "Del" button. The MenuButton defines the type of the Property. Click and hold it with the left mouse button and choose from the popup menu. The "Name:" text field can be edited by clicking it with the left mouse button. SHIFT-BACKSPACE clears the name.



Property names are case sensitive. So "Erwin" is not equal to "erwin".

The next field is different for each of the Property types. For the boolean type there are two radio-buttons; choose between "True" and "False". The string-type accepts a string; enter a string by clicking in the field with the left mouse. The other types are using a NumberButton to define the default value. Use SHIFT-LMB for editing it with the keyboard, click and drag to change the value with the mouse.

Property types

Boolean (Bool)

This Property type stores a binary value, meaning it can be "TRUE" or "FALSE". Be sure to write it all capital when using these values in Property Sensors or Expressions.

Integer (Int)

Stores a number like 1,2,3,4,... in the range from -2147483647 to 2147483647.

Float

Stores a floating point number.


String

Stores a text string. You can also use Expressions or the Property Sensor to compare strings.

Timer

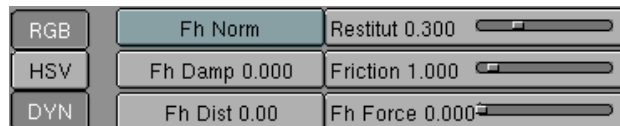
This Property type is updated with the actual game time in seconds, starting from zero. On newly created object the timer starts when the object is "born".

2.6. Settings in the MaterialButtons

Some physical attributes can be defined with the material settings of Blender. The MaterialButtons can be accessed via the  icon in the header of the ButtonsWindow or by pressing F5. Create a new material or choose an existing one with the MenuButton in the header.

In the MaterialButtons you need then to activate the "DYN" button to see the dynamic settings (See [Figure 2-6](#)).

Figure 2-6. Material settings for dynamic objects



Restitut

This parameter controls the elasticity of collisions. A value of 1.0 will convert all the kinetic energy of the object to the opposite force. This object then has an ideal elasticity. This means that if the other object (i.e. the ground) also has a Restitut of 1.0 the object will keep bouncing forever.

Friction

This value controls the friction of the objects material. If the friction is low, your object will slide like on ice, with a high friction you get an effect like sticking in glue.

Fh Force

In conjunction with the "Do Fh" and/or "Rot Fh" (see [Section 2.4](#)) you make an object float above a surface.

"Fh Force" controls the force that keeps the object above the floor.

Fh Dist

"Fh Dist" controls the size of the Fh area. When the object enters this area the Fh mechanism starts to work.

Fh Damp

Controls the damping inside the Fh area. Values above 0.0 will damp the object movement inside the Fh area.

Fh Norm

With this button activated the object also gets a force in the direction of the face normal on slopes. This will cause an object to slide down a slope (see the example: FhDemo.blend (blends/FhDemo.blend)).

2.6.1. Specularity settings for the gameEngine

Figure 2-7. Specularity settings



Specularity settings in the MaterialButtons

Spec

This slider controls the intensity of the specularity.

Hard

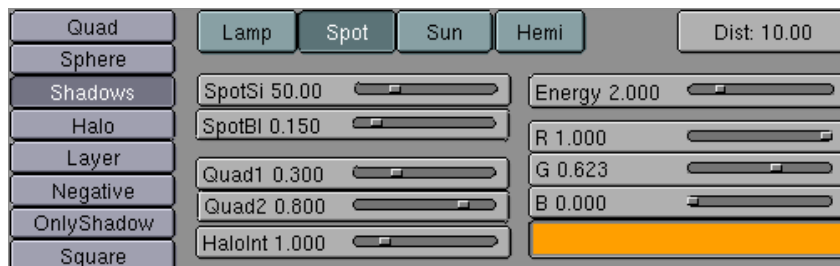
This slider controls the size of the specularity (hardness)

Spec color

Activating this button, switches the RGB (or HSV) sliders to define the specularity color

2.7. Lamps in the gameEngine


Figure 2-8. LampButtons, settings for gameBlender



Lamps are created with the Toolbox (SPACE->ADD Lamp). For a selected lamp you can switch to the LampButtons (F4) to change the properties of that lamp. These properties are the color, the energy,

etc. Due to the fact that the gameEngine is fully integrated in Blender, there are some buttons which are only useful for linear animation.

Common settings for all lamp types are the energy, and the color (adjustable with the RGB sliders).

To allow a face to receive realtime lighting in gameBlender the face has to be set to "Light" in the Paint/FaceButtons  (See [Chapter 4](#)). With the layer settings for lamps and objects (EditButtons, F9) you can control the lighting very precise. Lamps only affect faces on the same layer(s) as the lamp. Per Layer you can use eight lamps (OpenGL limitation) for realtime lighting.

Lamp types for the gameEngine

Lamp

Lamp is a point light source.

Spot

This lamp is restricted to a conical space. In the 3DWindow the form of the spotlight is shown with broken lines. Use the SpotSi slider to set the angle of the beam.

Sun

The "Sun" lamp type is a directional light. The distance has no effect on the intensity. Change the direction of the light (shown as a broken line) by rotating the lamp.

Hemi

"Hemi" lamp type is currently not supported in the gameEngine.

The "Lamp" and "Spot" lights can be sensitive to the distance. Use the "Dist:", "Quad1:" and "Quad2:" settings for this. The mathematics behind this are explained in the Official Blender 2.0 Guide (<http://www.blender.nl/show>

Load a demo file (blends/LampTypes.zip) showing the lamp types in the gameEngine. Press PKEY to launch the demo.

2.8. The blender laws of physics


All objects in Blender with the "Dynamic" option set (see [Settings for gameObjects](#)) are evaluated with the physics laws as defined by the gameEngine and the user.

The key property for a dynamic object is its mass. Gravity, forces, and impulses (collision bounce) only work on objects with a mass. Also, only dynamic objects can experience drag, or velocity damping (a crude way to mimic air/water resistance).




Note that for dynamic objects using dLoc and dRot may not have the desired result. Since the velocity of a dynamic object is controlled by the forces and impulses, any explicit change of position or orientation of an object may not correspond with the velocity. For dynamic objects it's better to use the linV and angV for explicitly defining the motion.

As soon we have defined a mass for our dynamic object it will be affected by gravity, causing it to fall until it hits another object with its bounding sphere. The size of the bounding-sphere can be changed with the "Size:" parameter in the RealtimeButtons. The gravity has a value of 9.81 by default: you can change this in the WorldButtons with the "Grav" slider. A gravity of zero is very useful for space games or simulations.

-  Use the "Damp:" and "RotDamp:" settings to suggest the drag of air or other environments. Don't use it to simulate friction. Friction can be simulated by using the dynamic material settings.

Dynamic objects can bounce for two reasons. Either you have Do Fh enabled and have too little damping, or you are using a Restitut value in the dynamic material properties that is too high.

-  If you haven't defined a material, the default restitution is 1.0, which is the maximum value and will cause two objects without materials to bounce forever.

In the first case, increasing the damping can decrease the amount of bounce. In the later case define a material for at least one of the colliding objects, and set its Restitut value to a smaller value. The Restitut value determines the elasticity of the material. A value of zero denotes that the relative velocity between the colliding objects will be fully absorbed. A value of one denotes that the total momentum will be preserved after the collision.

Damping is a decrease of velocity in % per second. Damping is useful to achieve a maximum speed. The larger the speed the greater the absolute decrease of speed due to drag. The maximum speed is attained when the acceleration due to forces equals the deceleration due to drag. Damping is also useful for damping out unwanted oscillations due to springs.

Friction is a force tangent to the contact surface. The friction force has a maximum that is linear to the normal, i.e., the force that presses the objects against each other, (the weight of the object). The Friction value denotes the Coulomb friction coefficient, i.e. the ratio of the maximum friction force and the normal force. A larger Friction value will allow for a larger maximum friction. For a sliding object the friction force will always be the maximum friction force. For a stationary object the friction force will cancel out any tangent force that is less than the maximum friction. If the tangent force is larger than the maximum friction than the object will start sliding.

For some objects you need to have different friction in different directions. For instance a skateboard will experience relatively little friction when moving it forward and backward, but a lot of friction when moving it side to side. This is called anisotropic friction. Selecting the "Anisotropic" button in the RealTimeButtons (F8) will enable anisotropic friction. After selecting this button, three sliders will appear in which the relative coefficient for each of the local axes can be set. A relative coefficient of zero denotes that along the corresponding axis no friction is experienced. A relative coefficient of one denotes that the full friction applies along the corresponding axis.

If you have suggestions or questions concerning the gameBlender physics please contact Gino (gino@blender.nl) (<mailto:gino@blender.nl>).

2.9. Expressions

Expressions can be used in the [Expression Controller](#), the [Property Sensor](#) and the [Property Actuator](#).

Table 2-1. Valid expressions

Expression type	Example
Integer numbers	15
Float number	12.23224
Booleans	TRUE, FALSE
Strings	"I am a string!"
Properties	propname
Sensornames	sensorname (as named in the LogicBrick)

Table 2-2. Arithmetic expressions

Expression	Example
EXPR1 + EXPR2	Addition, 12+3, propname+21
EXPR1 - EXPR2	Subtraction, 12-3, propname-21
EXPR1 * EXPR2	Multiplication, 12*3, propname*21
EXPR1 / EXPR2	Division, 12/3, propname/21
EXPR1 > EXPR2	EXPR1 greater EXPR2
EXPR1 >= EXPR2	EXPR1 greater or equal EXPR2
EXPR1 < EXPR2	EXPR1 less EXPR2

Table 2-3. Boolean operations

Operation	Example
NOT EXPR	Not EXPR
EXPR1 OR EXPR2	logical OR
EXPR1 AND EXPR2	logical AND
EXPR1 == EXPR2	EXPR1 equals EXPR2

Conditional statement: IF(Test, ValueTrue, ValueFalse)

Examples:

Table 2-4. Expression examples

Expression	Result	Explanation
12+12	24	Addition
property=="Carsten"	TRUE or FALSE	String comparison between a Property and a string
"Erwin">"Carsten"	TRUE	A string compare is done

2.10. SoundButtons


The SoundButtons  are for loading and managing sounds in gameBlender. Look at [Section 2.11](#) for a possibility to visualize the waveform.

Figure 2-9. The SoundButtons

In the SoundButtons header you can see the name of the SoundObject (here "SO:MiniGunFire.wav"). This name is set to the name of the sound sample by default. With the MenuButton you can browse existing SoundObjects and create new SoundObjects. The blue color of the sound name indicates that more than one user uses the sound, the number button indicates the number of users.

In the SoundButtons you can then assign or load samples in the SoundObject. So the SoundObject name doesn't have to be the name of the sample. For example you can use a SoundObject "SO:explosion" and then load "explosion_nuke.wav" later. You load samples with the "Load Sample" button in the SoundButtons. The sample name and the location on disk are shown in the text field to the right of the "Load Sample" button. With the MenuButton left to the location, you can browse already loaded samples and assign it to the SoundObject.

The NumberButton indicates how many SoundObjects share the sample. When the pack/unpack button (parcel) is pressed, the sample is packed into the *.blend file, which is especially important when distributing files.

The "Play" button obviously plays the sound, the "Loop" button set the looping for the sample on or off. Depending on the play-mode in the [Sound Actuator](#) this setting can be overridden.

The "Vol:" slider sets the global volume of the sound.

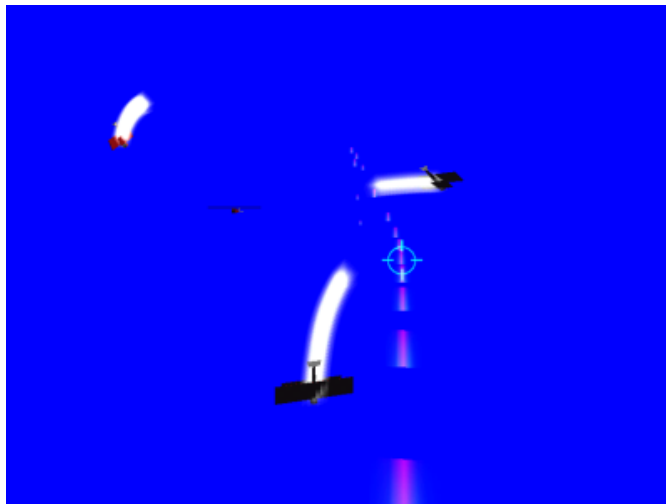
Pitch: with the pitch you can change the frequency of the sound. Currently there's support for values between half the pitch (-12 semitones) and double the pitch (+12 semitones). Or in Hertz: if your sample has a frequency of 1000 Hz, the bottom value is 500 and the top 2000 Hz.

The next row of buttons lets you define 3D sound. With the Button "3D vol" activated (which deactivates "Fixed") you enable 3D sound. This means the volume of the sound depends on the distance between the sound source and the listener. The listener is the active camera.

The "Attn:" slider sets the sound attenuation. In a 3D world you want to scale the relation between gain and distance. For example, if a sound passes by the camera you want to set the scaling factor that determines how much the sound will gain if it comes towards you and how much it will diminish if it goes away from you. Currently, the scaling factor can be set between 0.0 (all positions get multiplied by 0: no matter where the source is, it will always sound as if it was playing in front of you) and 1.0 (a neutral state, for all positions get multiplied by 1).

The next row of buttons defines the stereo position of the sound. With "3D pan" activated the volume for the left and right stereo channel of the sound is dependant of the position relative to the listener. When "Fixed" is activated you can manually pan the sound with the "Pann:" slider.

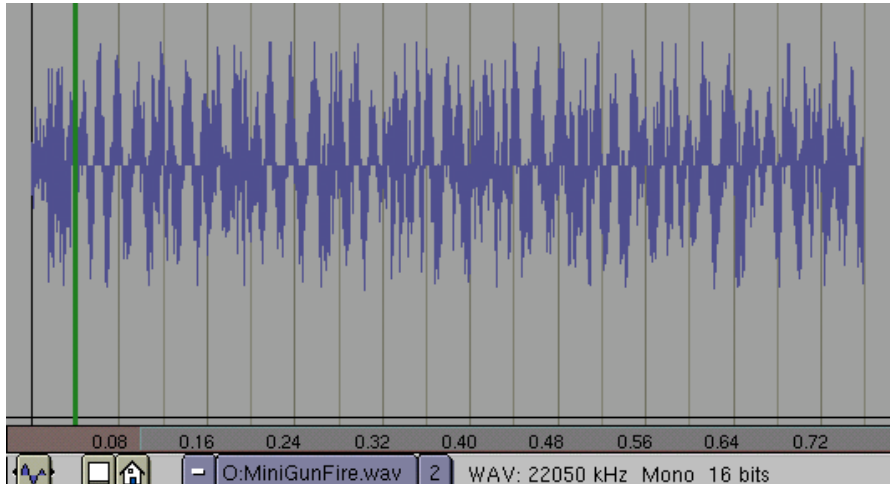
Figure 2-10. 3D sound demo



Get the 3D sound demo. (blends/MG1.blend)

2.11. SoundWindow

Figure 2-11. The SoundWindow



The SoundWindow is used to load and visualize sounds. You can grab and zoom the window and its content like every other window in Blender.

The green bar indicates the position of the FrameSlider. This can be used to synchronize a sound with an IPO animation. In the lower part of the window you also have an indicator of the sound length in seconds.

In the SoundWindow header see the usual window buttons, the user buttons and some information about the sound.

2.12. Performance and design style issues

Computers get faster every month, nearly every new computer nowadays has a hardware accelerated graphics card. But still there are some performance issues to think about. This is not only a good style in design and programming but also essential for the platform compatibility Blender provides. So to make a well designed game for the various platforms keep these rules in mind:

1. Originals for an AddObject Actuator must be in an invisible layer (very important, maybe we will even force this in future)
2. Don't use properties in combination with AND/OR/Expr. controller as scripting language. There is a python controller.
3. Don't share (Python) variables between scripts
4. As little inter-object LogicBrick connections as possible.

5. Use ALT-D (instanced mesh for new object) when replicating meshes, this is better than SHIFT-D (copies the mesh)
6. Alpha mapped polygons are expensive, so use with care
7. Switching off collision flag for polygons is good for performance, also the use of 'ghost' is cheaper than a regular physics object
8. Keep the polygon count as low as possible. It's quite easy to add polygons to models, but very hard to remove them without screwing up the model. The detail should be made with textures.
9. Keep your texture-resolution as low as possible. You can work with hi-res versions and then later reduce them for publishing the game (see Chapter [UV Texturing](#))
10. Polygons set to "Light" are expensive. A hardware acceleration with Transform and Lighting calculation will help here.
11. Instead of real-time lighting use VertexPaint to lighten, darken or tint faces to suggest light situations.

Chapter 3. Game LogicBricks

The game logic in gameBlender is assembled in the RealtimeButtons. Here you wire the different LogicBricks together. The following is a brief documentation on all the LogicBricks currently available.

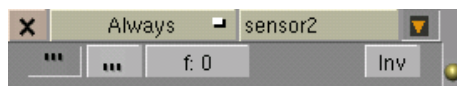
3.1. Sensors

Sensors act like real senses; they can detect collisions, feel (Touch), smell (Near), view (Ray, Radar).

3.1.1. Always Sensor

The most basic Sensor is the Always Sensor. It is also a good example for the common buttons every sensor has.

Figure 3-1. Common elements for Sensors



The button labeled "X" deletes the Sensor from the gameLogic. This happens without a confirmation, so be careful. The MenuButton right to the delete button (here labeled "Always") allows you to choose the type of Sensor. Click and hold it with the left mouse button to get the popup menu. Next is a TextButton, which holds the name of the Sensor. Blender assigns the name automatically on creation. Click the name with the left mouse button to change the name with the keyboard.



Name your LogicBricks and Blender objects to keep track of your scenes. A graphical logic scheme can become very complex.

With the small arrow button you can hide the contents of the LogicBrick, so it only shows the top bar. This is very handy in complex scenes.

The next row of buttons is used to determine how and at which frequency a Sensor is "firing". This topic is a bit complex, so we will give examples in more than one part of this documentation.

General things on pulses

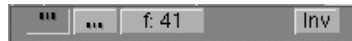
Pulses coming from Sensors trigger both Controllers and Actuators. A pulse can have two values, TRUE or FALSE.

Each Controller is always evaluated when it receives a pulse, whether the pulse is TRUE or FALSE doesn't matter. The input 'gate' of a Controller remembers the last pulse value. This is necessary for Controllers being linked by multiple Sensors, then it can still do a logical AND or OR operation on

all inputs. When a Controller is triggered, and after evaluation of all inputs, it can either decide to execute the internal script or to send a pulse to the Actuators.

An Actuator reacts to a pulse in a different way, with a TRUE pulse it switches itself ON (makes itself active), with a FALSE pulse it turns itself OFF.

Figure 3-2. Pulse Mode Buttons



The first button activates the positive pulse mode. Every time the Sensor fires a pulse it is a positive pulse. This can be used, for example to start a movement with an Motion Actuator. The button next to it activates the negative pulse mode, which can be used to stop a movement.



If none of the pulse mode buttons are activated the Always Sensor fires exactly one time. This is very useful for initialising stuff on game start.

The button labeled "f:" (set to 41 here), determines the delay between two pulses fired by the Sensor. The value of "f:" is given as frames.

The "Inv" button inverts the pulse, so a positive (TRUE) pulse will become negative (FALSE) and vice versa.

Get a example file here:[pulses.blend](#) (blends/LogicBricks212/1.sensors/pulses.blend).

3.1.2. Keyboard Sensor

The Keyboard Sensor is maybe one of the most often used Sensors because it provides the interface between Blender and the user (we will cover other methods like the mouse later).

The pulse mode buttons are common for every Sensor so they have the same functionality as described for the Always sensor.

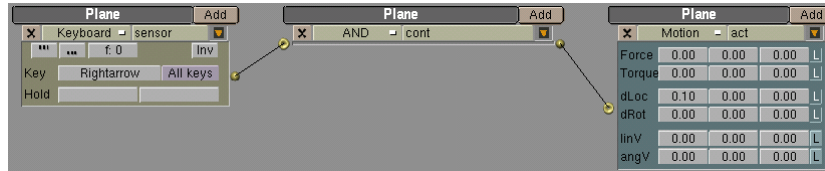


Clear Blender with CTRL-X to the default scene. The default plane is active and selected (pink). Switch to the RealTimeButtons with F8. Use the "Add"-Buttons to add one LogicBrick to the Sensor, Controller and Actuator blocks.

Change the Always Sensor with the MenuButton to a Keyboard Sensor. Then click on the "Key" input field and press the key you want to assign.

Now connect the LogicBricks. Click and hold the left mouse on the yellow ball at a LogicBrick and drag the appearing line to the yellow donut of the next LogicBrick.

Figure 3-3. LogicBricks for moving the default plane



Change the first dLoc field (they are ordered x, y, z) to 0.10. Move your mousecursor over the big 3DWindow and press PKEY to start the gameEngine. Now press the key you assigned to the Keyboard Sensor and the plane should move to the right. A more complex example can be found in the LogicBricks examples (blends/LogicBricks212.zip), keyboard.blend (blends/LogicBricks212/1.sensors/keyboard

By activating the "All keys" Button, the Sensor will react on every key. In the "Hold" fields you can put in modifier keys, which need to be held while pressing the main key.

Python methods:

```
setKey( int key );
```

Sets the Key on which the Sensor reacts

```
int key getKey( );
```

Gets the key on which the Sensor reacts

```
setHold1( int key );
```

Sets the modifier key one

```
int key getHold1( );
```

Gets the modifier key one

```
setHold2( int key );
```

Sets the modifier key two

```
int key getHold2( );
```

Gets the modifier key two

```
setUseAllKeys( bool all );
```

Sets the "All keys" option on all=TRUE

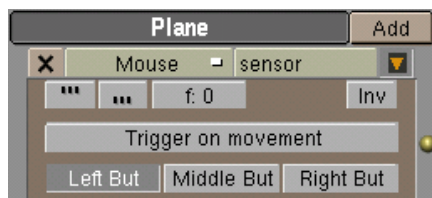
```
bool all getUseAllKeys( );  
    Gets the state of the "All keys" option
```

```
list keys getPressedKeys( );  
    Gets a list of the pressed keys
```

```
list events getKeyEvents( );  
    Gets a list of the keys events
```

3.1.3. Mouse Sensor

The Mouse Sensor includes the usual buttons like every sensor. The main purpose is of course to react mouse input from on the user. Curenly the Sensor is able to watch for mouse clicks or mouse movement. To get the position of the mouse cursor as well you need to use a Python-script right now. This will be covered later in this documentation (see). Get an example from the LogicBricks examples (blends/LogicBricks212.zip), mouse_button.blend (blends/LogicBricks212/1.sensors/mouse_button.blend).



Python methods:

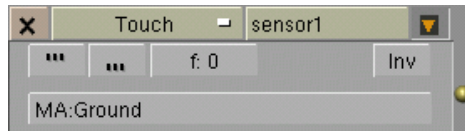
```
int xposgetXPosition( );  
    Gets the mouse x-position
```

```
int yposgetYPosition( );  
    Gets the mouse y-position
```

3.1.4. Touch Sensor

The Touch Sensor fires a pulse when the object it is assigned to, touches a material. If you enter a material name into the "MA:" text field it only reacts on this material otherwise it reacts on all touches.

This way you can achieve effects like a lava material causing the player to die when he touches it.



Python methods:

```
setTouchMaterial( (char* matname) );
```

Sets the Material the Touch Sensor should react on

```
char* matname getTouchMaterial( );
```

Gets the Material the Touch Sensor reacts on

```
gameObject obj getHitObject( );
```

Returns the touched Object

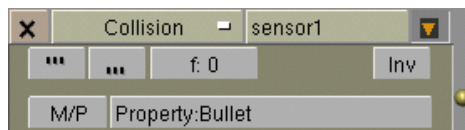
```
list objs getHitObjectList( );
```

Returns a list of touched objects

You find an example file in the LogicBricks examples (blends/LogicBricks212.zip), touch.blend (blends/LogicBricks212

3.1.5. Collison Sensor

The Collision Sensor is a general Sensor to detect contact between objects. Besides reacting on materials it is also capable of detecting Properties of an object. Therefore you can switch the input field from Material to Property bit clicking on the "M/P" button.



Python methods:

```
setTouchMaterial( (char* matname) );
```

Sets the Material the Collision Sensor should react on

```
char* matname getTouchMaterial( );
```

Gets the Material the Collision Sensor reacts on

```
setProperty( (char* propName) );
```

Sets the Property the Collision Sensor should react on

```
char* propName getProperty( );
```

Gets the Property the Collision Sensor reacts on

```
gameObject obj getHitObject( );
```

Returns the colliding Object

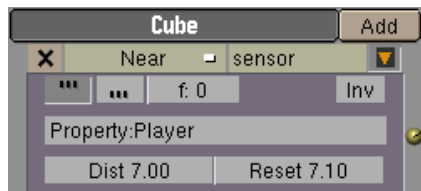
```
list objs getHitObjectList( );
```

Returns a list of collided objects

For an example look at the LogicBricks examples (blends/LogicBricks212.zip), collision.blend (blends/LogicBricks212/

3.1.6. Near Sensor

The near sensor reacts on actors near the object with the sensor.

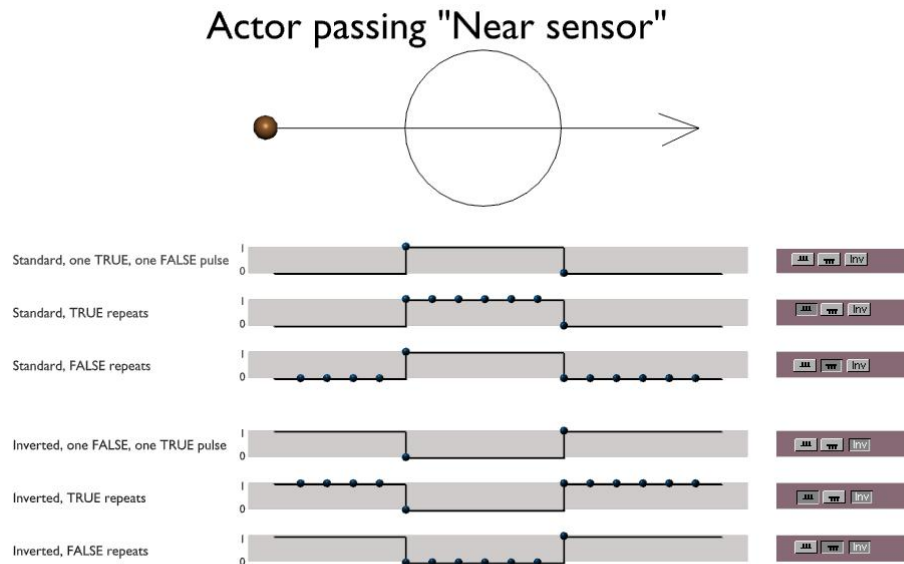


i The near sensor only senses objects of the type "Actor" (a dynamic object is also an actor).

If the "Property:" field is empty, the near sensor reacts on all actors in its range. If filled with a property name, the sensor only reacts on actors carrying a property with that name.

The range (spherical) of the near sensor you set with the "Dist" NumberButton. The "Reset" value defines at what distance the near sensor is reset again. This is helpful to avoid multiple pulses when an actor is just at the "Dist" distance, or to open a door at a certain distance but close it on a different distance.

Figure 3-4. Schematic of pulses for a near sensor



Python methods:

```
setProperty( (char* propname) );
```

Sets the Property the Near Sensor should react on

```
char* propname getProperty( );
```

Gets the Property the Near Sensor reacts on

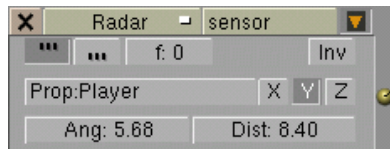
More to come...

Examples for the use of the Near Sensor are an enemy reaction when the Player gets close, a door opening, etc.

For an example file look at the LogicBricks examples (blends/LogicBricks212.zip), near_property.blend (blends/LogicBricks212/1.sensors/near_property.blend).

3.1.7. Radar Sensor

The Radar Sensor acts like a real radar. It looks for an object along the axis indicated with the axis buttons "X, Y, Z". If a property is entered into the "Prop:" field, it only reacts on objects with this property.



In the "Ang:" field you can enter an opening angle for the radar. This equals angle of view for a camera. The "Dist:" setting determines how far the Radar Sensor can see.

Objects can't block the line of sight for the Radar Sensor. This is different for the Ray Sensor (see [Section 3.1.10](#)). You can combine them for making a radar that's not able to look through walls.

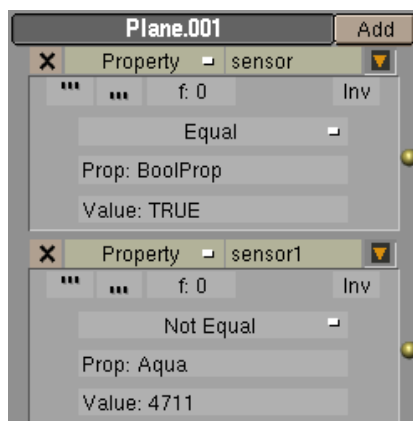
Examples:

Logic Bricks examples (blends/LogicBricks212.zip).

radar3.blend (blends/LogicBricks212/1.sensors/radar3.blend).

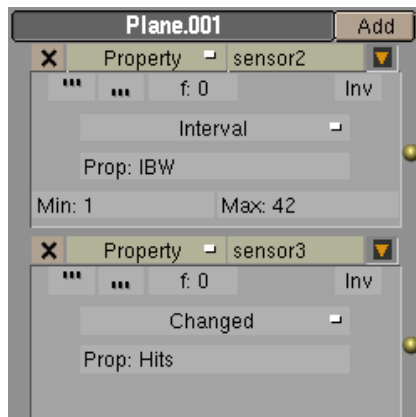
3.1.8. Property Sensor

The Property Sensor logically checks a Property attached to the same object.



The property sensor of the type "Equal" checks for equality of the property given in the "Prop:" field and the value in "Value:". If the condition is true, it fires pulses according to the pulse mode settings.

The "Not Equal" checks for inequality and then fires its pulses.



The "Interval" type property sensor fires its pulse if the value of property is inside the interval defined by "Min:" and "Max:". This sensor type is especially helpful for checking float values, which you can't depend on to reach a value exactly. This is most common with the "Timer" Property.

The "Changed" Property Sensor gives pulses every time a Property is changed. This can for example happen through a Property Actuator, a Python script or an Expression.

Python methods:

```
setProperty( (char* propname) );
```

Sets the Property to check

```
char* propname getProperty( );
```

Gets the Property to check

```
setType( (int type) );
```

Sets the type of the Property Sensor.

1. Equal
2. Not Equal
3. Interval
4. Changed

```
char* propname getProperty( );
```

Gets the type of the Property Sensor

```
setValue( (char* expression) );
```

Sets the value to check (as expression)

```
char* expression getValue( );
```

Gets the value to check (as expression)

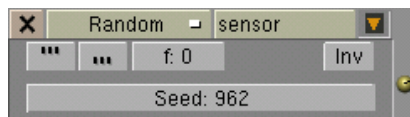
Examples:

Logic Bricks examples (blends/LogicBricks212.zip).

property.blend (blends/LogicBricks212/1.sensors/property.blend).

3.1.9. Random Sensor

The Random Sensor fires a pulse randomly according to the pulse settings (50/50 pick).



With a seed of zero the Random Sensor works like an Always Sensor, which means it fires a pulse every time.

Python methods:

```
setSeed( (int seed) );
```

Set the seed for the random generation

```
int seed getSeed( );
```

Gets the seed for the Random Sensor

```
int seed getLastDraw( );
```

Gets the last draw from the Random Sensor


Examples:

LogicBricks examples (blends/LogicBricks212.zip).

random.blend (blends/LogicBricks212/1.sensors/random.blend).

3.1.10. Ray Sensor

The Ray Sensor casts a ray for the distance set into the NumberButton "Range". If the ray hits an object with the right Property or the right Material the Sensor fires its Pulse.

 Other objects block the ray, so it can't see through walls.



Without a material or property name filled in, the Ray Sensor reacts on all objects.

Python methods:

```
list [x,y,z] getHitPosition( );
```

Returns the position where the ray hits the object.

```
list [x,y,z] getHitNormal( );
```

Returns the normal vector how the ray hits the object.

```
list [x,y,z] getRayDirection( );
```

Returns the vector of the Ray direction

```
gameObject obj getHitObject( );
```

Returns the hit Object

Examples:

Logic Bricks examples (blends/LogicBricks212.zip).

ray.blend (blends/LogicBricks212/1.sensors/ray.blend).

Known issues:

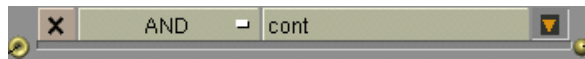
Only Y-Axis works for now, as workaround use an Empty to carry the Ray Sensor and rotate it in such a way that the Y-Axis points along the desired direction.

If the Ray Sensor refuses to work with a Material as trigger check if the latest object created has the desired material, if not assign it to it (perhaps you need to make a helper object).

3.2. Controllers

Controllers act as the brain for your game logic. This reaches from very simple decisions like connecting two inputs, over slightly complex expressions, to complex Python scripts which can carry artificial intelligence.

3.2.1. AND Controller



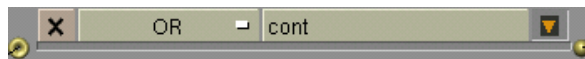
The AND Controller combines one, two or more inputs from Sensors. That means that all inputs must be active to pass the AND Controller.

Examples:

LogicBricks examples (blends/LogicBricks212.zip).

and.blend (blends/LogicBricks212/2.controllers/and.blend).

3.2.2. OR Controller



The OR Controller combines one, two or more inputs from Sensors. OR means that either one or more inputs can be active to let the OR Controller pass the pulse through.

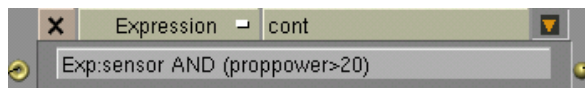
Examples:

LogicBricks examples (blends/LogicBricks212.zip).

or.blend (blends/LogicBricks212/2.controllers/or.blend).

3.2.3. Expression Controller

With the Expression Controller you can create slightly complex game logic with a single line of 'code'. You can access the output of sensors attached to the controller and access the properties of the object.



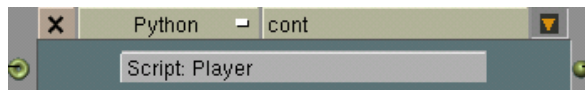
The expression mechanism prints out errors to the console or in the DOS window, so have a look there if anything fails.

Examples:

LogicBricks examples (blends/LogicBricks212.zip).

expression.blend (blends/LogicBricks212/2.controllers/expression.blend).

More on using Expressions can be found in [Section 2.9](#).

3.2.4. Python Controller

The Python controller is the most powerful controller in gameBlender. You can attach a Python script to it, which allows you to control your gameObjects ranging from simple movement up to complex gameplay and artificial intelligence.

Enter the name of the script you want to attach to the Python Controller into the "Script:" field. The script needs to be existant in the scene or Blender will ignore the name you type.



Remember that Blender treats names case sensitive! So a script "player" is not the same as "Player".

Python for the gameEngine is covered in Chapter [Section 5.2](#)

Python methods:

```
Actuator* getActuator( char* name , );
```

Returns the actuator with "name".

```
list getActuators( );
```

Returns a python list of all connected Actuators.

```
Sensor* getSensor( char* name , );
```

Returns the Sensor with "name".

```
list getSensors( );
```

Returns a python list of all connected Sensors.

Examples:

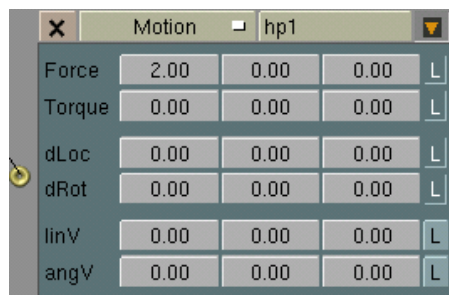
LogicBricks examples (blends/LogicBricks212.zip).

Python examples (blends/LogicBricks212/2.controllers/).

3.3. Actuators

Actuators are the executing LogicBricks. They can be compared with muscles or glands in a life form.

3.3.1. Motion Actuator



The Motion Actuator is surely the most important Actuator. It moves, rotates or applies a velocity to objects.

The simplest case of using a Motion Actuator is to move the object. This is done with the "dLoc" values in the third row. Every time the actuator is triggered by an impulse it moves the object the amount given in the "dLoc" row. The three values here stand for X-, Y- and Z-axis. So when you enter a 1.0 in the first field the object is moved one unit per time unit of the game (the clock in the gameEngine ticks in frames, roughly 1/25 of a second, for exact timings use the Time Property). A simple example is given in the [Keyboard Sensor](#) section.

The buttons labeled "L" behind each row in the motion actuator, determine if the motion applied should be treated as global or local. If the button is pushed (dark green) the motion is applied based on the local axis of the object. If the button is not pressed the motion is applied based on the global (world) axis.

Force

Values in this row act as forces that apply to the object. This works only for dynamic objects.

Torque

Values in this row act as rotational forces (Torque) that apply to the object. This works only for dynamic objects. Positive values rotate counter-clock-wise.

dLoc

Offset the object as indicated in the value fields

dRot

Rotate the object for the given angle (36 is a full rotation). Positive values rotate clock-wise.

linV

Sets the velocity of the object to the given values.

angV

Sets the angular velocity to the given values. Positive values rotate counter-clock-wise

The Motion Actuator starts to move objects on a pulse (TRUE) and stops it on a FALSE pulse. To get a movement over a certain distance, you need to send a FALSE pulse to the motion actuator always. See the demo file "blends/MoveCertainDist.blend" (blends/MoveCertainDist.blend) for an example.

Python methods:

```
setForce( list [x,y,z] , bool local );
```

Set the "Force" parameter for the Motion Actuator... ???

```
list [x,y,z] getForce( );
```

Gets the "Force" parameter for the Motion Actuator.

```
setTorque( list [x,y,z] );
```

Set the "Torque" parameter for the Motion Actuator

```
list [x,y,z] getTorque( );
```

Gets the "Torque" parameter for the Motion Actuator

```
setdLoc( list [x,y,z] );
```

Sets the dLoc parameters from the Motion Actuator

```
list [x,y,z] getdLoc( );
```

Get the dLoc parameters from the Motion Actuator

```
setdRot( list [x,y,z] );
```

Sets the dRot parameters for the Motion Actuator

```
list [x,y,z] getdLoc( );
```

Gets the dRot parameters from the Motion Actuator

```
setLinearVelocity( list [x,y,z] );
```

Sets the linV parameters for the Motion Actuator

```
list [x,y,z] getLinearVelocity( );
```

Gets the linV parameters from the Motion Actuator

```
setAngularVelocity( list [x,y,z] );
```

Sets the angV parameters for the Motion Actuator

```
list [x,y,z] getAngularVelocity( );
```

Gets the angV parameters from the Motion Actuator

Examples:

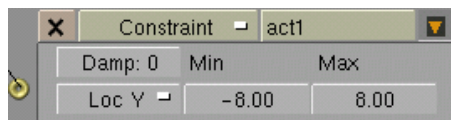
LogicBricks examples (blends/LogicBricks212.zip).

motion.blend (blends/LogicBricks212/3.actuators/motion.blend).

motion_LinV.blend (blends/LogicBricks212/3.actuators/motion_LinV.blend).

3.3.2. Constraint Actuator

With the Constraint Actuator you can limit an objects freedom to a certain degree.



With the MenuButton you specify the channel of which freedom should be constrained. With the NumberButtons "Min" and "Max" you define the minimum and maximum values for the constraint selected. To constrain an object on more than one channel simply use more than one Constraint actuator.

Examples:

LogicBricks examples (blends/LogicBricks212.zip).

Python methods:

```
setDamp( int damp );
```



```
int damp getDamp( );
```

```
setMin( int min );
```

```
int min getMin( );
```

```
setMax( int max );
```

```
int max getMax( );
```

```
setMin( int min );
```

```
int min getMin( );
```

```
setMin( int min );
```

```
int min getMin( );
```

```
setLimit( ? limit );
```

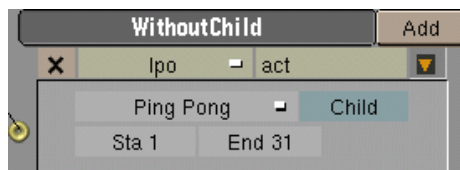
```
int limit getLimit( );
```

Known issues:

Damping is not working in Blender 2.12

Constraint for rotation not implemented in 2.12

3.3.3. IPO Actuator



The IPO Actuator can play the IPO-curves for the object that owns the Actuator. If the object has a child with an IPO (in a parenting chain) and you activate "Child" in the Actuator, the IPO for the child is also played.

IPO play modes

Play

Plays the IPO from "Sta" to "End" at every positive pulse the Actuator gets. Another pulse while playing is discarded.

Ping Pong

Plays the IPO from "Sta" to "End" on the first positive pulse, then backwards from "End" to "Sta" when the second positive pulse is received.

Flipper

Plays the IPO as long as the pulse is positive. When the pulse changes to negative the IPO is played from the current frame to "Sta".

Loop Stop

Plays the IPO in a loop as long as the pulse is positive. It stops at the current position when the pulse turns negative.

Loop End

This plays the IPO repeatedly as long as there is a positive pulse. When the pulse stops it continues to play the IPO to the end and then stops.

Property

Plays the IPO for exactly the frame indicated in the property entered in the field "Prop:".

Currently following IPOs are supported by the gameEngine:

Mesh Objects

Loc, Rot, Size and Col

Lamps

Loc, Rot, RGB, Energy

Cameras

Loc, Rot, Lens, ClipSta, ClipEnd

Python methods:

```
setType( ??? , );
```

```
int type GetType( ??? , );
```

```
SetStart( int frame , );
```

```
SetEnd( int frame , );
```

```
int frameGetStart( );
```

```
int frameGetEnd( );
```

Examples:

LogicBricks examples (blends/LogicBricks212.zip).

mouse_button.blend (blends/LogicBricks212/1.sensors/mouse_button.blend).

3.3.4. Camera Actuator



The Camera Actuator tries to mimic a real cameraman. It keeps the actor in the field of view and tries to stay at a certain distance from the object. Also the motions is soft and there is some delay in the reaction on the motion of the object.

Fill in the object that should be followed by the camera (you can also use the Camera Actuator for non-camera objects) into the "OB:" field. The field "Height:" determines the height the camera stays above the object. "Min:" and "Max:" are the bounds of the distance from the object in which the camera is allowed to move. The "X" and "Y" buttons specify behind which axis of the object the camera tries to stay.

Examples:

Logic Bricks examples (blends/LogicBricks212.zip).

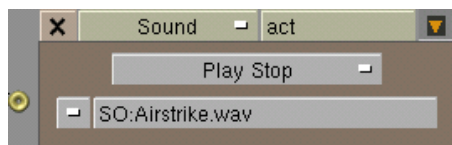
camera.blend (blends/LogicBricks212/3.actuators/camera.blend).

Known issues:

2.12: Visibility check not implemented, so an obstacle can cover the object.

camera.blend (blends/LogicBricks212/3.actuators/camera.blend).

3.3.5. Sound Actuator



The Sound Actuator plays a SoundObject loaded with the SoundButtons (see [Section 2.10](#)).

Sound play modes

Play Stop

Plays the sound as long as there is a positive pulse.

Play End

Plays the sound to the end, when a positive pulse is given.

Loop Stop

Plays and repeats the sound, when a positive pulse is given.

Loop End

Plays the sound repeatedly, when a positive pulse is given. When the pulse stops the sound is played to its end.

Examples:

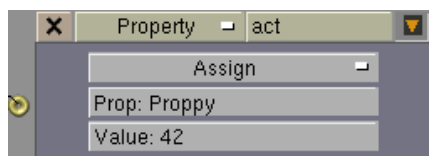
LogicBricks examples (blends/LogicBricks212.zip).

sound_property.blend (blends/LogicBricks212/3.actuators/sound_property.blend).

Known issues:

2.12: Sound only on Windows and Linux implemented

3.3.6. Property Actuator



Property modes

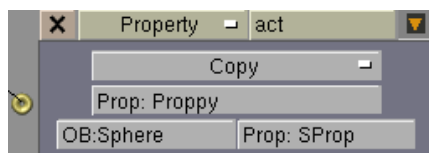
Assign

Assigns a value or **Expression** (given in the "Value" field) to a Property. For example with an Expression like "Proppy + 1" the "Assign" works like an "Add". To assign strings you need to add quotes to the string ("...").

Add

Adds the value or result of an expression to a property. To subtract simply give a negative number in the "Value:" field.

Copy



This copies a Property (here "Prop: SProp") from the Object with the name given in "OB: Sphere" into the Property "Prop: Proppy". This is an easy and safe way to pass information between objects.

Python methods:

```
SetProperty( char* name );
```

```
*char name GetProperty( );
```

```
SetValue( char* value );
```

```
char* value GetValue( );
```

Examples:

LogicBricks examples (blends/LogicBricks212.zip).

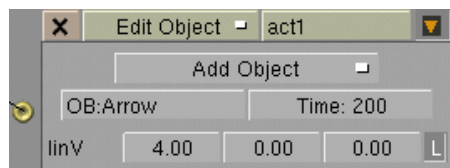
More on using Expressions can be found in [Section 2.9](#).

3.3.7. Edit Object Actuator

This actuator performs actions on Objects itself, like adding new objects, deleting objects, etc.

Edit Object Actuator types

Add Object



The Add Object actuator adds an object to the scene. The new object is oriented along the x-axis of the creating object.



Keep the object you like to add on a separate and hidden layer or it will not work as expected.

Enter the name of the Object to add in the "OB:" field. The "Time:" field determines how long (in frames) the object should exist. The value "0" denotes it will exist forever. Be careful not to slow down gameBlender by generating too many objects! If the time an object should exist is not

predictable, you can also use other events (collisions, properties, etc.) to trigger an "End Object" for the added object by using LogicBricks.

With the "linV" buttons it is possible to assign an initial velocity to the added object. This velocity is given in x, y and z components. The "L" button stands for local. When it is pressed the velocity is interpreted as local to the added object.

Python methods:

```
setObject( char* name );
```

Sets the Object to be added

```
char* name getObject( char* name );
```

Gets the Object name

```
setTime( int time );
```

Time in frames the added Object should exist. Zero means unlimited

```
int time getTime( );
```

Gets the time the added object should exist

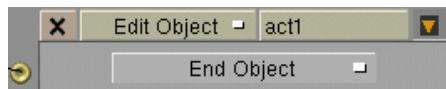
```
setLinearVelocity( list [vx,vy,vz] );
```

Sets the linear velocity [Blenderunits/sec] components for added Objects.

```
list [vx,vy,vz] getLinearVelocity( );
```

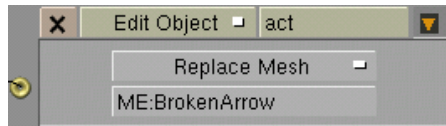
Sets the linear velocity [Blenderunits/sec] components from the Actuator

End Object



The "End Object" type simply ends the life of the object with the actuator when it gets a pulse. This is very useful for ending a bullet's life after a collision or something similar.

Replace Mesh



The "Replace Mesh" type, replaces the mesh of the object by a new one, given in the "ME:" field. Remember that the mesh name is not implicitly equal to the objectname.

Python methods:

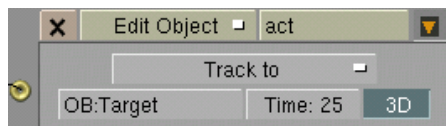
```
setMesh( char* name );
```

Sets the Mesh for the ReplaceMesh Actuator to "name"

```
char* name getMesh( );
```

Gets the Mesh-name from the ReplaceMesh actuator

Track to



The "Track to" type, rotates the object in a way that the y-axis points to the target specified in the "OB:" field. Normally this happens only in the x/y plane of the object (indicated by the "3D" button not pressed). With "3D" pressed the tracking is done in 3D. The "Time:" parameter sets how fast the tracking is done. Zero means immediately, values above zero give a delay (slower) in tracking.

Python methods:

```
setObject( char* name );
```

```
char* name getObject( );
```

```
setTime( int time );
```



```
int time getTime( );
```

```
setUse3D( );
```

```
bool 3d setUse3D( );
```

Examples:

LogicBricks examples (blends/LogicBricks212.zip).

end_object2.blend (blends/LogicBricks212/3.actuators/end_object2.blend).

add_object2.blend (blends/LogicBricks212/3.actuators/add_object2.blend).

3.3.8. Scene Actuator

The Scene Actuator is meant for switching Scenes and Cameras in the gameEngine.

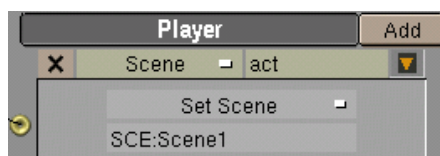
Choose the desired action with the MenuButton and enter an existing camera or scene name into the text field. If the name does not exist, the button will be blanked!

Reset



Simply restarts and resets the scene. It has the same effect like stopping the game with ESC and restart with PKEY.

Set Scene



Switch to the scene indicated into the text field. During the switch all properties are reset!

Python methods:

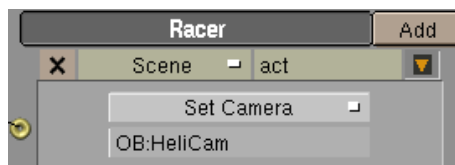
```
setScene( char* scene );
```

Sets the Scene to switch to

```
char* scene getScene( );
```

Gets the Scenename from the Actuator

Set Camera



Switch to the Camera indicated into the text field.

Python methods:

```
setCamera( char* camera );
```

Sets the Camera to switch to

```
char* camera getCamera( );
```

Gets the Camera name from the Actuator

Examples:

SetScene.blend (blends/SetScene.blend).

SetCamera.blend (blends/SetCamera.blend).

3.3.9. Random Actuator

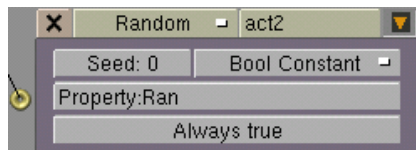
An often-needed function for games is a random value to get more variation in movements or enemy behavior.

The Seed parameter is the value fed into the random generator as a start value for the random number generation. Because computer generated random numbers are only "pseudo" random (they will repeat after a (long) while) you can get the same random numbers again if you choose the same Seed.

Fill the name of the property you want to be filled with the random number into the "Property:" field.

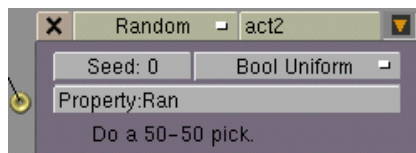
Random Actuators types

Boolean Constant



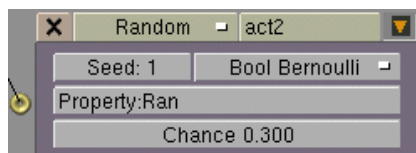
This is not a random function at all, use this type to test your game logic with a TRUE or FALSE value.

Boolean Uniform



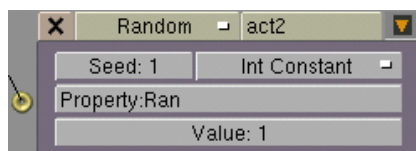
This is the classical random 50-50 pick. It results in TRUE or FALSE with the same chance. This is like a (ideal) coin-pick.

Boolean Bernoulli



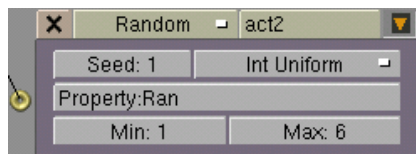
This random function results also in a boolean value of TRUE or FALSE, but instead of having the same chance for both values you can control the chance of having a TRUE pick with the "Chance" parameter. A chance of 0.5 will be the same as "Bool Uniform". A chance of 0.1 will result in 1 out of 10 cases in a TRUE (on average).

Integer Constant



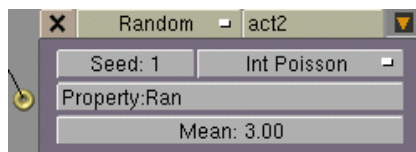
For testing your logic with a value given in the "Value:" field

Integer Uniform



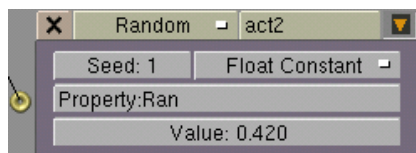
This random type randomly produces an integer value between (and including) "Min:" and "Max:". The classical use for it is to simulate a dice pick with "Min: 1" and "Max: 6".

Integer Poisson



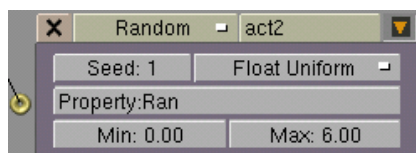
The random numbers are distributed in such a way that an average of "Mean:" is reached with an infinite number of picks.

Float Constant



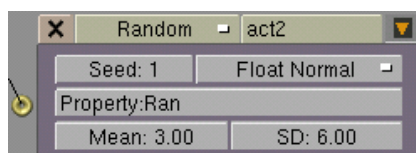
For debugging your game logic with a given value.

Float Uniform



This returns a random floating point value between "Min:" and "Max:".

Float Normal



Returns a weighted random number around "Mean:" and with a standard deviation of "SD:".

Float Negative Exponential



Returns a random number which is well suited to describe natural processes like radioactive decay or lifetimes of bacteria. The "Half-life time:" sets the average value of this distribution.

Python methods:

```
setSeed( int seed );
```

Sets the random seed (the init value of the random generation)

```
int seed getSeed( );
```

Gets the random seed (the init value of the random generation) from the Actuator

```
float para1 getPara1( );
```

Gets the first parameter for the selected random distribution

```
float para2 getPara2( );
```

Gets the second parameter for the selected random distribution

```
setProperty( char* proptype );
```

Sets the to which Property the random value should go

```
char* proptype getProperty( );
```

Gets the Property name from the Actuator

```
setDistribution( int dist );
```

??

```
int dist getDistribution( );
```

Gets the random distribution method from the Actuator

Chapter 4. UV Texturing

Textures have a big impact on the look and feel of your game or interactive environment. With textures you are able to create a very detailed look even with a low poly model. With alpha channel textures you are also able to create things like windows or fences without actually modeling them.

4.1. Available file formats

Blender uses OpenGL (<http://www.opengl.org/>) to draw its interface and the gameEngine. This way we can provide the great multi platform compatibility. In terms of using textures we have to pay attention to several things before we're able to run the game on every Blender platform.

- the height and width of textures need to be of a power of 64 pixels (e.g. 64x64, 64x128, 128x64 etc.)
- it is not recommended to use textures with a resolution above 256x256, because not all graphic cards support higher resolutions.

Blender can use the following file formats as (real-time) textures:

Targa

The Targa or TGA (*.tga extension) file format is a lossless compressed format, which can include an alpha channel

Iris

Iris (*.rgb) is the native IRIX image format. It is a lossless compressed file format, which can include an alpha channel.

Jpeg

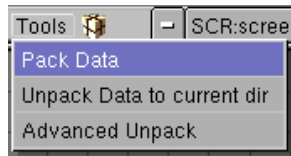
A lossy compressed (it uses a compression which leaves out parts in the image which the human eye can hardly see) file format (*.jpg, *.jpeg) designed for photos with very small file sizes. Because of its small footprint it is a very good format for distribution over the net. It has no support for alpha channels and is because of the quality loss due to compression not a recommended as format to work with during the design phase of a game.

4.2. Handling of resources

For publishing and easier handling of Blenders files, you can include all resources into the scene. Normally textures, samples and fonts are not included in a file while saving. This keeps them on your

disk and makes it possible to change them and share between scenes. But if you want to distribute a file it is possible to pack these resources into the Blendfile, so you only need to distribute one file, preventing missing resources.

Figure 4-1. The ToolsMenu



The functions for packing and unpacking are summarized in the ToolsMenu. You can see if a file is packed if there is a little "parcel" icon right to the ToolsMenu. After you packed a file, all new added resources are automatically packed (AutoPack).

The Tools Menu entries

Pack Data

This packs all resources into the Blendfile. The next save will write the packed file to disk.

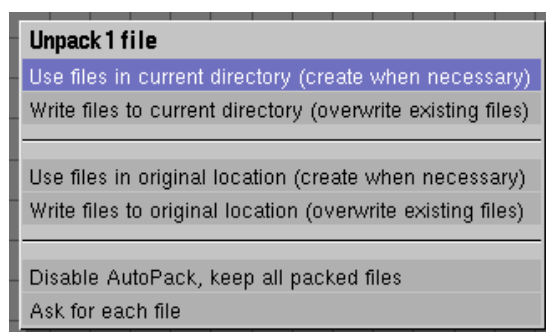
Unpack Data to current dir

This unpacks all resources to the current directory. For textures a directory "textures" is created, for sounds a "samples" directory and fonts are unpacked to "fonts".

Advanced Unpack

This option calls the Advanced Unpack Menu.

Figure 4-2. Advanced Unpack Menu



Advanced Unpack Menu entries

Use files in current directory

This unpacks only files which are not present in the current directory. It creates files when necessary.

Write files to current directory

This unpacks the files to the current directory. It overwrites existing files!

Use files in original location

This uses files from their original location (path on disk). It creates files when necessary.

Write files to original location

This writes the files to their original location (path on disk). It overwrites existing files!

Disable AutoPack, keep all packed files

This disables AutoPack, so new inserted resources are not packed into the Blendfile.

Ask for each file

This asks the user for each file for the unpack options.

4.3. The UV Editor

The UV editor is fully integrated into Blender and allows you to map textures onto the faces of your models. Each face can have individual texture coordinates and an individual image assigned. This can be combined with vertexcolors to darken or lighten the texture or to tint it.

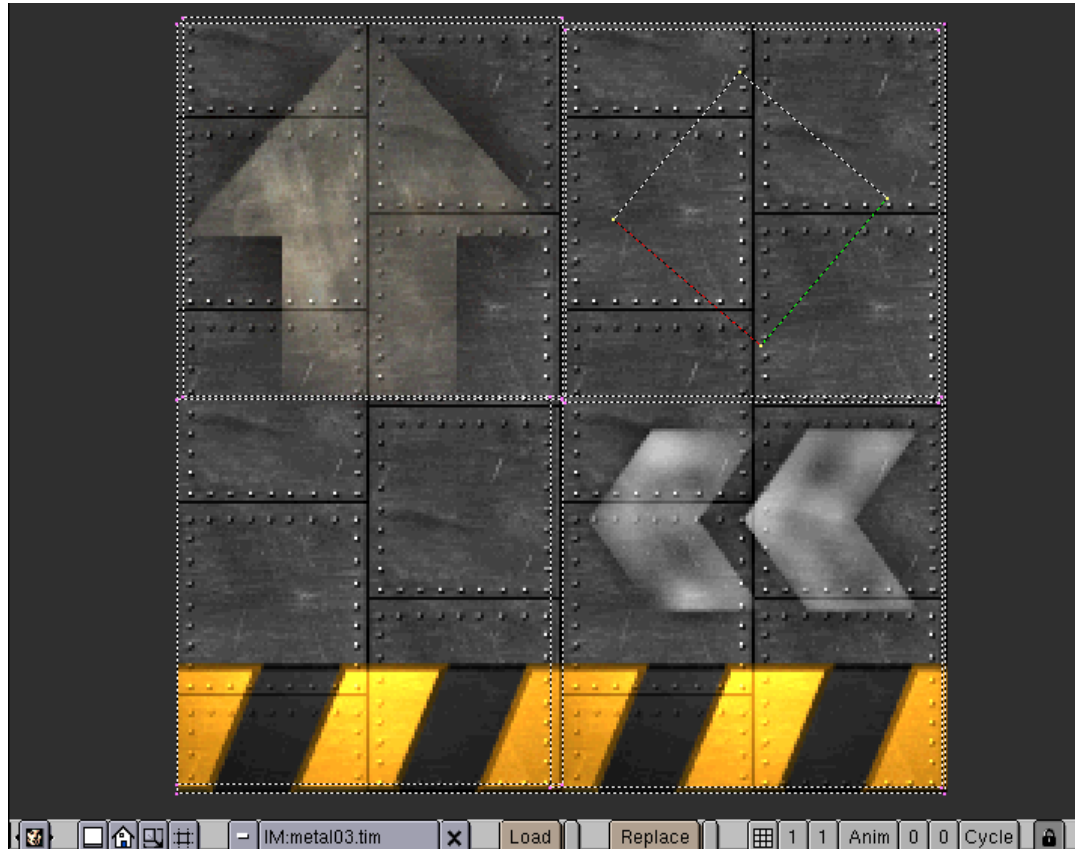
To start UV editing, enter FaceSelect mode with the FKEY or the FaceSelect icon in the 3DWindow header. The mesh is now drawn z-Buffered. In textured mode (ALT-Z) untextured faces are drawn purple to indicate the lack of a texture. Selected faces are drawn with a dotted outline.

To select faces use the right mouse button, with the BKEY you can use BorderSelect and the AKEY selects/deselects all faces. While in FaceSelect mode you can enter EditMode (TAB) and select vertices. After leaving EditMode the faces defined by the selected vertices are selected in FaceSelect mode. The active face is the last selected face: this is the reference face for copy options.

4.3.1. The ImageWindow

To assign images to faces you need to open an ImageWindow with SHIFT-F5.

Figure 4-3. The Image Window



The first Icon keeps UV polygons square while editing: this is a big help while texturing. Just drag one or two vertices around and the others are following to keep the polygon square. The second one keeps the vertices inside the area of the image.



With the UserBrowse (MenuButton) you can browse, assign and delete loaded images on the selected faces.

"Load" loads a new image and assigns it to the selected faces. "Replace" replaces (scene global) an image on all faces assigned to the old image. The small buttons right to the "Load" and "Replace" buttons opens a FileWindow without the thumbnail images.



The grid icon enables the use of more (rectangular) images in one map. This is used for texturing from textures containing more than one image in a grid and for animated textures. The following two number buttons define how many parts the texture has in x and y direction. Use SHIFT-LMB to select the desired part of the image in GridMode.

The "Anim" button enables a simple texture animation. This works in conjunction with the grid settings, in a way that the parts of the texture are displayed in a row in game mode. With the number buttons right of the "Anim" button you define the start and end part to be played. "Cycle" switches between one-time and cyclic play.

With the lock icon activated, any changes on the UV polygons in the ImageWindow are shown in realtime in the 3DWindows (in textured mode).

Vertices in the ImageWindow are selected and edited (rotate, grab) like vertices in EditMode in the 3DWindows. Drag the view with the middle mouse, zoom with PAD+ and PAD-.

4.3.2. The Paint/FaceButtons


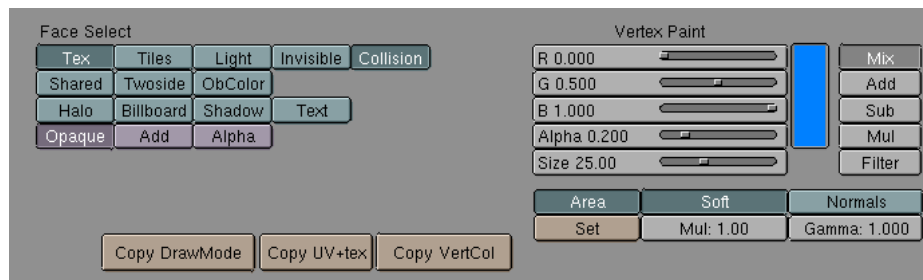
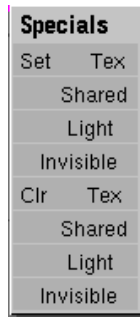
When in FaceSelect mode, you can access the Paint/FaceButtons with the Icon  in the ButtonsWindow header. In the Paint/FaceButtons you'll find all functions to set the attributes for faces and access the VertexPaint options.

Figure 4-4. The Paint/FaceButtons



The following modes always work on faces and display the setting of the active face. Two colored lines in the 3D and the ImageWindow indicate the active face. The green line indicates the U coordinate, the red line the V coordinate. To copy the mode from the active to the selected faces use the copy buttons ("Copy DrawMode", "Copy UV+tex" and "Copy VertCol") in the Paint/FaceButtons. In FaceSelect mode the special menu has some points to quickly set and clear modes on all selected faces, see [Figure 4-5](#).

Figure 4-5. The special menu for the FaceSelectMode



Face modes

Tex

This enables the use of textures. To use objects without textures disable "Tex" and paint the faces with VertexPaint.

Tiles

This indicates and sets the use of the tile mode for the texture, see [Section 4.3.1](#).

Light

Enables realtime lighting on faces. Lamps only affect faces of objects in the same layer as the lamp. Lamps can also be on more than one layer, which makes it possible to create complex real-time lighting situations. See also [Section 2.7](#).

Invisible

Makes faces invisible. These faces are still calculated for collisions, so this gives you an option to build invisible barriers, etc.

Collision

The faces with this option are evaluated by the gameEngine. If that is not needed, switch off this option to save on resources.

Shared

With this option vertex colors are blended across faces if they share vertices (currently not implemented in Blender 2.12).

Twoside

Faces with this attribute are rendered twosided in the gameEngine

ObColor

Faces can have an animatable color using the ColR, ColG and ColB IPOs. This option replaces the vertexcolors

Halo

Faces with this attribute are rendered with the X-axis always pointing to the active view or camera.

Billboard

Faces with this attribute are pointing in the direction of the active view with the X-axis. The difference with "Halo" is the faces are only rotated around the Z-axis

Shadow

(currently not implemented in Blender 2.12).

Text

Faces with this attribute are used for displaying bitmap-text in the gameEngine, see [Section 4.3.3](#).

Opaque

Normal opaque rendered faces. The color of the texture is rendered as color.

Add

Faces are rendered transparent. The color of the face is added to what has already been drawn. Black areas in the texture are transparent, white are fully bright. Use this option to achieve light beam effects, glows or halos around lights. For real transparency use the next option.

Alpha


Depending on the alpha channel, the face is rendered transparent.

4.3.3. Bitmap text in the game engine

Since Blender version 2.04 we've got the ability to draw text in the gameEngine using bitmap fonts. These bitmap fonts can be created from a TrueType or Postscript outline font. For an explanation how to create a bitmap font look for the Tutorial How to create your own bitmap fonts (<http://www.blender.nl/showitem.php?id=42>) on the Blender site.

To get bitmap text or numbers displayed on a single face you need a special bitmap with the font rendered onto it. Then create a property named "Text" for your object and map the first character of the text-bitmap on it. Check the "Text" face attribute for the face, which should display the text in the Paint/FaceButtons. The type of the property can be every type, so also a boolean will be rendered as "True" or "False". For a complete tutorial look at: Using bitmap fonts in blender 2.04 (<http://www.blender.nl/showitem.php?id=42>)

Chapter 5. Python

 Work in progress! Especially I will add here more Python script examples in future.

Python (www.python.org) is an interpreted, interactive, object-oriented programming language.

Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. Python is also usable as an extension language for applications that need a programmable interface.

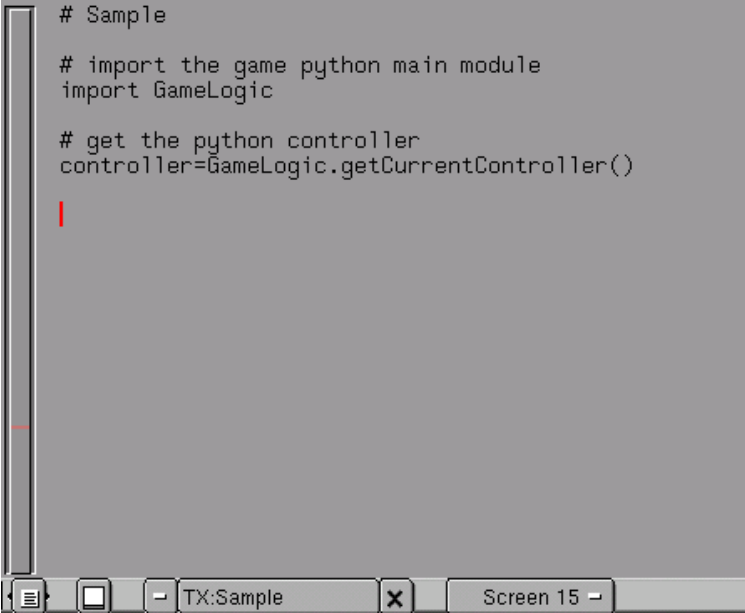
Beside this use as extension language, the Python implementation is portable to (at least) all platforms that Blender runs on.

Python is copyrighted but freely usable and distributable, even for commercial use.

5.1. The TextWindow

The TextWindow is a simple but useful text editor, fully integrated into Blender. It's main purpose of it is to write Python scripts, but it is also very useful for writing comments in the Blendfile or to explain other users the purpose of the scene.

Figure 5-1. The TextWindow



```
# Sample
# import the game python main module
import GameLogic

# get the python controller
controller=GameLogic.getCurrentController()

|
```

The TextWindow can be displayed with SHIFT-F11 or by adjusting the IconMenu in the Window-header. As usual there is an IconBut to make the TextWindow fullscreen, the next MenuButton can be

used to switch between text files, open new ones or add new text buffers. The x-shaped Button deletes a textbuffer after a confirmation.

With the right MenuButton you can change the font for displaying the text.

With LeftMouse-Hold and dragging the mouse you can mark ranges of text for the usual cut, copy & paste functions. The keycommands are:

Keycommands for the TextWindow

ALT-C

Copy the marked text into a buffer

ALT-X

Cut out the marked text into a buffer

ALT-V

Paste the text from buffer to the cursor in the TextWindow

ALT-O

Loads a text, a FileWindow appears

CTRL-R

Reloads the current text, very useful for editing with an external editor

SHIFT-ALT-F

Pops up the Filemenu for the TextWindow

ALT-J

Pops up a NumButton where you can specify a linenumber the cursor will jump to

ALT-U

Unlimited Undo for the TextWindow

ALT-R

Redo function, recovers the last Undo

ALT-A

Mark the whole text

5.2. Python for games

With Python integrated into the gameEngine you can influence LogicBricks, change their parameters and react on events triggered by the LogicBricks.

Beside that you can influence the `gameObject` that carries the Python Controller directly. This means moving it, applying forces or getting information from this object.

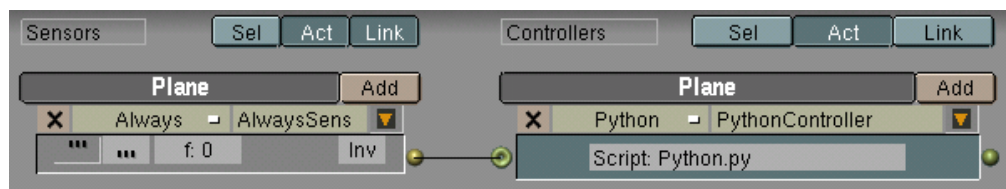


Beside the Python in the gameEngine, Blender also includes Python for modeling and animation tasks.

5.2.1. Basic gamePython

The first step for using gamePython is to add at least a Sensor and a Python Controller to an object. Then add a new text file in the TextWindow. Fill in the name of that text file into the "Script:" field of the Python Controller. You should now have a gameLogic setup like in [Figure 5-2](#).

Figure 5-2. LogicBricks for a first gamePython script.



Now enter the following script (`./blends/Python_getPosition.blend`) into the TextWindow you just created (you don't need to type the lines starting with "#", these are comments).

Figure 5-3. First Script

```

1      # first gamePython script
2      # gets the position of the owning object
3      # and prints it on the console
4
5      import GameLogic
6
7      controller = GameLogic.getCurrentController()
8      owner = controller.getOwner()
9
10     print owner.getPosition()
```

The "print" command and errors from the Python interpreter will appear on the console where you started Blender from, or in the DOS window, when running Blender under Windows. So it is helpful to size the Blender window in such a way that you can see the console window while programming Python.

This basic script only prints the position of the object that owns the Python Controller. Move your object and then restart gameBlender with the PKEY to see the results changing.

Now to explain the function of the script line by line. Line five is maybe the most important line here. We import the GameLogic module which is the basis for all gamePython in Blender.

In line seven we get the Controller, which executes the script and assigns it to the variable "controller".

In line eight we use the controller we got in line seven to get the owner, the GameObject carrying the LogicBrick. You can see we use the method "getOwner()" to get the owner of our controller.

We now have the owner and we can use its methods to do things with it. Here in line 10 we use the "getPosition()" method to print the position of the gameObject as a matrix of the X, Y and Z values.

You may now wonder what other methods the PythonObjects have. Of course this is part of this documentation, but Python is "self" documenting, so we have more possibilities to get that information.

Add the following line to the end of the script from [Figure 5-3](#):

```
1 print dir(owner)
```

Start the gameEngine again, stop it and look at the console window. You see the following output:

```
[0.0, 0.0, 0.0]
['applyImpulse', 'disableRigidBody', 'enableRigidBody', 'getLinearVelocity', 'getMass',
'getOrientation', 'getPosition', 'getReactionForce', 'getVelocity', 'restoreDynamics', 'setOrientation',
'setPosition', 'setVisible', 'suspendDynamics']
```

The first line shows the position of the object, the next lines show the methods, which the "owner" is providing. For example you see a 'getMass' method, which will return the mass of a dynamic object. With the knowledge of the "dir()" function you can ask Python objects for information, without consulting external documentation.

5.3. Examples for gamePython

Here you will find some small samples for the use of Python in the gameEngine. Please explore the provided Blend-files and re-use them in your own scenes.

5.3.1. Simple visibility check

Figure 5-4. Visibility Script

```
1         # Simple visibility check
2
3         import Rasterizer
4
5
6         # experimental visibility flag
7         # it will hide all objects
```

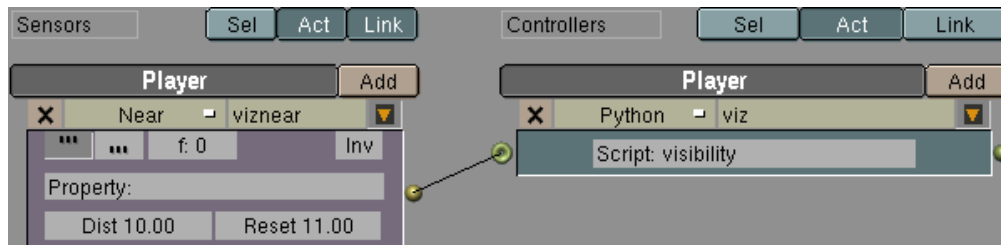
```

8      # so objects must be set visible
9      # using this script
10
11     Rasterizer.enableVisibility(1)
12
13
14     # get controller and sensor
15     cont = GameLogic.getCurrentController()
16     sensor = cont.getSensor("viznear")
17
18     # Set the player visible ;- )
19     sensor.getOwner().setVisible(1)
20
21
22     # now set all objects in the range of the near
23     # visible. Try to change "Dist" and "Reset"
24     objects = sensor.getHitObjectList()
25     if objects:
26         for obj in objects:
27             obj.setVisible(1)

```

For a simple visibility check you assign a Near Sensor to your player as shown in [Figure 5-5](#). The Near Sensor will provide the visibility script with an object list that is inside the Near Sensor range defined by "Dist" and "Reset". Make sure you check the positive pulse mode and set "f:0".

Figure 5-5. LogicBricks for the visibility script



Get a example file here: [Visibility.blend](#) (blends/Visibility.blend).

5.3.2. Mouse cursor with python

Figure 5-6. Mouse cursor script

```

1      # Mouse cursor control
2

```

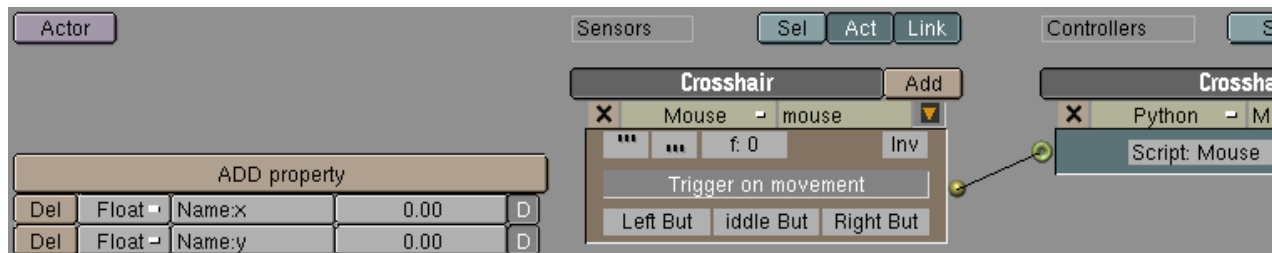
```

3     import GameLogic
4     import Rasterizer
5
6     # Get the dimensions of the game window
7     ww = Rasterizer.getWindowWidth()
8     wh = Rasterizer.getWindowHeight()
9
10    # calculate the aspect (in full screen the aspect of the CRT)
11    aspect = float(wh)/ww
12
13    # With this value you can calibrate the mouse to the 3D Window
14    actionx=28.0
15    actiony=actionx*aspect
16
17    #Get controller, sensor and owner
18    c = GameLogic.getCurrentController()
19    sensor = c.getSensor("mouse")
20    owner = c.getOwner()
21
22    # I use properties for debugging purpose (use wireframe to see)
23    owner.x = float(sensor.getXPosition()-ww/2)/ww*actionx
24    owner.y = float(sensor.getYPosition()-wh/2)/wh*actiony
25
26    # Set the position of the crosshair directly
27    # (see SCA, there is no Actuator)
28    owner.setPosition([owner.x,0,-owner.y])

```

The script is triggered by a Mouse Sensor, set to "Trigger on movement". Note the use of the two Properties, x and y, they are used for debugging but needed for the script to work. If you don't need the Properties replace the occurrence of "owner.x" with "x" and "owner.y" with "y".

Figure 5-7. Logic bricks for the mouse script



Get a example file here:MG1.blend (blends/MG1.blend).

5.4. gamePython Documentation per module

5.4.1. GameLogic Module

```
SCA_PythonController getCurrentController( );
```

Returns the Controller object which carries the script.

```
void addActiveActuator( actuator , bool active );
```

This method makes the Actuator "actuator" active ("active=TRUE") or inactive ("active=FALSE").

```
float getRandomFloat( );
```

This function returns a random float in the range of 0.0...1.0. The seed is taken from the system time, so you get a different sequence of random numbers at every game start.

```
setGravity( [gravityX,gravityY,gravityZ] );
```

Sets the world gravity.

5.4.2. Rasterizer Module

```
int getWindowWidth( );
```

This function returns the width of the Blenderwindow the game is running in.

```
int getWindowHeight( );
```

This function returns the height of the Blenderwindow the game is running in.

```
void makeScreenshot( char* filename );
```

This function writes a screenshot of the game as TGA file to disk.

```
enableVisibility( bool usevisibility );
```

This sets all objects to invisible when "usevisibility" is TRUE. The game can then set the visibility back to on for only the needed objects.

```
showMouse( bool show );
```

Shows the mouse cursor when "show" is TRUE while the gameEngine runs.

```
setBackgroundColor( [float R,float G,float B] );
```

```
setMistColor( [float R,float G,float B] );
```

```
setMistStart( float start );
```

```
setMistEnd( float end );
```

5.4.3. GameKeys Module

This is a module that only defines all keyboard keynames (AKEY = 65 etc).

"AKEY", ..., "ZKEY", "ZERO_KEY" , ..., "NINEKEY", "CAPSLOCKKEY", "LEFTCTRLKEY", "LEFTALTKEY", "RIGHTALTKEY", "RIGHTCTRLKEY", "RIGHTSHIFTKEY", "LEFTSHIFTKEY", "ESCKEY", "TABKEY", "RETKY", "SPACEKEY", "LINEFEEDKEY", "BACKSPACEKEY", "DELKEY", "SEMICOLONKEY", "PERIODKEY", "COMMAKEY", "QUOTEKEY", "ACCENTGRAVEKEY", "MINUSKEY", "VIRGULEKEY", "SLASHKEY", "BACKSLASHKEY", "EQUALKEY", "LEFTBRACKETKEY", "RIGHTBRACKETKEY", "LEFTARROWKEY", "DOWNARROWKEY", "RIGHTARROWKEY", "UPARROWKEY", "PAD0", ..., "PAD9", "PADPERIOD", "PADVIRGULEKEY", "PADASTERKEY", "PADMINUS", "PADENTER", "PADPLUSKEY", "F1KEY", ..., "F12KEY", "PAUSEKEY", "INSERTKEY", "HOMEKEY", "PAGEUPKEY", "PAGEDOWNKEY", and "ENDKEY".

5.5. Standard methods for LogicBricks

All LogicBricks inherit the following methods:

```
getOwner( );
```

This returns the owner, the gameObject has the LogicBrick assigned.

```
setExecutePriority( int pri );
```

?

```
int pri getExecutePriority( );
?
```

5.5.1. Standard methods for Sensors

All sensors inherit the following methods:

```
int isPositive( );
```

1 if the sensor fires a positive pulse.

```
bool getUsePosPulseMode( );
```

Returns TRUE if positive pulse mode is active, FALSE if positive pulse mode is not active.

```
setUsePosPulseMode( bool flag );
```

Set flag to TRUE to switch on positive pulse mode, FALSE to switch off positive pulse mode.

```
int getPosFrequency( );
```

Return the frequency of the updates in positive pulse mode.

```
setPosFrequency( int freq );
```

Set the frequency of the updates in positive pulse mode. If the frequency is negative, it is set to 0.

```
bool getUseNegPulseMode( );
```

Returns TRUE if negative pulse mode is active, FALSE if negative pulse mode is not active.

```
setUseNegPulseMode( bool flag );
```

Set flag to TRUE to switch on negative pulse mode, FALSE to switch off negative pulse mode.

```
int getNegFrequency( );
```

Return the frequency of the updates in negative pulse mode.

```
setNegFrequency( int freq );
```

Set the frequency of the updates in negative pulse mode. If the frequency is negative, it is set to 0.

```
bool getInvert( );
```

Returns whether or not pulses from this sensor are inverted.

```
setInvert( bool flag );
```

Set to TRUE to invert the responses of this sensor, set to FALSE to keep the normal response.

5.5.2. Standard methods for Controllers

Controllers have the following methods:

```
Actuator* getActuator( char* name , );
```

Returns the actuator with "name".

```
list getActuators( );
```

Returns a python list of all connected Actuators.

```
Sensor* getSensor( char* name , );
```

Returns the Sensor with "name".

```
list getSensors( );
```

Returns a python list of all connected Sensors.

5.5.3. Standard methods for gameObjects

GameObjects you got with `getOwner()` have the following methods.

```
applyImpulse( list [x,y,z] , );
```

Apply impulse to the gameObject (N*s).

```
disableRigidBody( );
```

Disables the rigid body dynamics for the gameObject.

```
enableRigidBody( , );
```

Enables the rigid body dynamics for the gameObject.

```
setVisible( int visible );
```

Sets the gameObject to visible.


```
setPosition( [x,y,z] );
```

Sets the position of the gameObject according to the list of the X, Y and Z coordinate.

```
pylist [x,y,z] getPosition( );
```

Gets the position of the gameObject as list of the X, Y and Z coordinate.

```
pylist [x,y,z] getLinearVelocity( );
```

Returns a list of the X, Y and Z component of the speed. The speed is in Blenderunits/second.

```
pylist [x,y,z] getVelocity( );
```

```
float massgetMass( );
```

Returns the mass of the gameObject.

```
pylist [x,y,z] getReactionForce( );
```

Returns a Python list of three elements.

```
suspendDynamics( );
```

Suspends the dynamic calculation in the gameEngine.

```
restoreDynamics( );
```

Suspends the dynamic calculation in the gameEngine.

Index

- Actuator
 - Camera, [52](#)
 - Constraint, [48](#)
 - Edit Object, [54](#)
 - IPO, [50](#)
 - Motion, [46](#)
 - Property, [53](#)
 - Random, [58](#)
 - Scene, [57](#)
 - Sound, [52](#)
- Actuators, [46](#)
- Add Object, [54](#)
- Always, [33](#)
- Anim, [67](#)
- animated textures, [67](#)
- AutoPack, [64](#)
- Bugs
 - , [52](#), [53](#)
 - Constraint, [50](#)
 - Ray Sensor, [43](#)
- Collision Sensor, [37](#)
- Command line options, [18](#)
- Controllers, [44](#)
 - AND, [44](#)
 - Expression, [44](#)
 - OR, [44](#)
 - Standard Methods, [80](#)
- Design style, [31](#)
- Examples, [40](#), [43](#), [45](#), [48](#), [48](#), [51](#), [52](#), [53](#), [54](#), [57](#)
 - AND Controller, [44](#)
 - Expressions, [28](#)
 - OR Controller, [44](#)
 - Phyton Controller, [45](#)
 - Property Sensor, [42](#)
 - Random Sensor, [42](#)
 - Scene Actuator, [58](#)
- Expressions, [28](#)
- Face modes, [68](#)
- file formats, [63](#)
- gameBlender, [9](#)
- GameKeys Module, [78](#)
- GameLogic Module, [77](#)
- GameMenu, [17](#)
- gameObjects
 - Standard Methods, [80](#)
- gravity, [27](#)
- grid icon, [67](#)
- ImageWindow, [65](#)
- Ketsji, [17](#)
- Keyboard, [34](#)
- Lamps, [25](#)
- Lights, [25](#)
- lock icon, [67](#)
- LogicBricks, [33](#)
 - Standard Methods, [78](#)
- Material, [25](#)
- Materials, [24](#)
- Mouse Sensor, [36](#)
- Near Sensor, [38](#)
- OpenGL, [26](#)
- Pack Data, [63](#)
- Paint/FaceButtons, [67](#)
- Performance, [31](#)
- Physics, [26](#)
- Properties, [22](#)
- property
 - Collision Sensor, [37](#)
 - Property Sensor, [40](#)
- Pulse, [33](#)
 - mode, [34](#)
- Python, [71](#), [72](#)
 - mouse, [75](#)
 - visibility, [74](#)
- Python methods
 - Property Actuator, [54](#)
- Python Controller, [45](#)
- Python examples, [74](#)
- Python methods
 - AddObject Actuator, [55](#)
 - Collision Sensor, [37](#)
 - Constraint Actuator, [48](#)
 - IPO Actuator, [51](#)
 - Motion Actuator, [47](#)
 - Mouse Sensor, [36](#)
 - Near Sensor, [39](#)

- Property Sensor, [41](#)
- Python Controller, [45](#)
- Random Actuator, [61](#)
- Random Sensor, [42](#)
- Ray Sensor, [43](#)
- ReplaceMesh Actuator, [56](#)
- Sensor, [35](#)
- SetCamera Actuator, [58](#)
- SetScene Actuator, [58](#)
- Touch Sensor, [37](#)
- TrackTo Actuator, [56](#)
- Python modules, [77](#)
- Radar Sensor, [40](#)
- Random Sensor, [42](#)
- Rasterizer Module, [77](#)
- Ray Sensor, [43](#)
- RealtimeButtons, [20](#)
- Resources, [63](#)
- Sensors, [33](#)
 - Standard Methods, [79](#)
- SoundButtons, [29](#)
- SoundWindow, [31](#)
- Special menu
 - FaceSelectMode, [67](#)
- Specularity, [25](#)
- Touch Sensor, [36](#)
- UV Editor, [65](#)
- UV Texturing, [63](#)
- WorldButtons, [27](#)

