12.8 Game Engine - Python API

Python API....1
Introduction....1
Bullet physics Python API....1
The VideoTexture module: bge.texture....2
How it works....2
Game preparation....3
First example....3
Find material....3
Create texture....4
Make texture persistent....4
Create a source....4
Setup the source....5
Play the video....5
Checking video status....6
Advanced work flow....6
Advanced demos....7

Python API

- Introduction
- Bullet physics Python API
- The VideoTexture module: bge.texture
    - How it works
    - Game preparation
    - First example
    - Advanced demos

Introduction

This site is currently under development.

To see the full Python API please click on the following link: Python API.

More informations:

- *Bullet physics*
- *Video Texture*

Bullet physics Python API

Bullet Physics provides collision detection and rigid body dynamics for the Blender Game Engine. It takes some settings from Blender that previously were designed for the former collision detection system (called

Sumo).

However, new features don't have an user interface yet, so Python can be used to fill the gap for now.

Features:

- Vehicle simulation.
- Rigid body constraints: hinge and point to point (ball socket).
- Access to internal physics settings, like deactivation time, debugging features.

Easiest is to look at the Bullet physics demos, how to use them. More information can be found here.

Python script example:

```
import PhysicsConstraints
print dir(PhysicsConstraints)
```

Note

Note about parameter settings

Since this API is not well documented, it can be unclear what kind of values to use for setting parameters. In general, damping settings should be in the range of 0 to 1 and stiffness settings should not be much higher than about 10.

The VideoTexture module: bge.texture

The `bge.texture` module allows you to manipulate textures during the game. Several sources for texture are possible: video files, image files, video capture, memory buffer, camera render or a mix of that. The video and image files can be loaded from the Internet using an URL instead of a file name. In addition, you can apply filters on the images before sending them to the GPU, allowing video effect: blue screen, color band, gray, normal map. `bge.texture` uses FFmpeg to load images and videos. All the formats and codecs that FFmpeg supports are supported by `bge.texture`, including but not limited to:

- AVI
- Ogg
- Xvid
- Theora
- dv1394 camera
- video4linux capture card (this includes many webcams)
- videoForWindows capture card (this includes many webcams)
- JPG

How it works

The principle is simple: first you identify an existing texture by object and name, then you create a new texture with dynamic content and swap the two textures in the GPU. The GE is not aware of the substitution and continues to display the object as always, except that you are now in control of the texture. At the end, the new texture is deleted and the old texture restored.

The present page is a guide to the `bge.texture` module with simple examples.

Game preparation

Before you can use the thing `bge.texture` module, you must have objects with textures applied appropriately.

Imagine you want to have a television showing live broadcast programs in the game. You will create a television object and UV-apply a different texture at the place of the screen, for example `tv.png`. What this texture looks like is not important; probably you want to make it dark grey to simulate power-off state. When the television must be turned on, you create a dynamic texture from a video capture card and use it instead of `tv.png`: the TV screen will come to life.

You have two ways to define textures that `bge.texture` can grab:

- Simple UV texture.
- Blender material with image texture channel.

Because `bge.texture` works at texture level, it is compatible with all GE fancy texturing features: GLSL, multi-texture, custom shaders, etc.

First example

Let’s assume that we have a game object with one or more faces assigned to a material/image on which we want to display a video.

The first step is to create a `Texture` object. We will do it in a script that runs once. It can be at the start of the game, the video is only played when you refresh the texture; we’ll come to that later. The script is normally attached to the object on which we want to display the video so that we can easily retrieve the object reference:

```
import bge.texture

contr = GameLogic.getCurrentController()
obj = contr.owner

if not hasattr(GameLogic, 'video'):
```

The check on `video` attribute is just a trick to make sure we create the texture only once.

Find material

```
matID = bge.texture.materialID(obj, 'IMvideo.png')
```

`bge.texture.materialID()` is a handy function to retrieve the object material that is using `video.png` as texture. This method will work with Blender material and UV texture. In case of UV texture, it grabs the internal material corresponding to the faces that are assigned to this texture. In case of Blender material, it grabs the material that has an image texture channel matching the name as first channel.

The `IM` prefix indicates that we're searching for a texture name but we can also search for a material by giving the `MA` prefix. For example, if we want to find the material called `VideoMat` on this object, the code becomes:

```
matID = bge.texture.materialID(obj, 'MAVideoMat')
```

Create texture

`bge.texture.Texture` is the class that creates the `Texture` object that loads the dynamic texture on the GPU. The constructor takes one mandatory and three optional arguments:

`gameObj`
: The game object.

`materialID`
: Material index as returned by `bge.texture.materialID()`, 0 = first material by default.

`textureID`
: Texture index in case of multi-texture channel, 0 = first channel by default. In case of UV texture, this parameter should always be 0.

`textureObj`
: Reference to another `Texture` object of which we want to reuse the texture. If we use this argument, we should not create any source on this texture and there is no need to refresh it either: the other `Texture` object will provide the texture for both materials/textures.

```
GameLogic.video = bge.texture.Texture(obj, matID)
```

Make texture persistent

Note that we have assigned the object to a `GameLogic`, `video` attribute that we create for the occasion. The reason is that the `Texture` object must be persistent across the game scripts. A local variable would be deleted at the end of the script and the GPU texture deleted at the same time. `GameLogic` module object is a handy place to store persistent objects.

Create a source

Now we have a `Texture` object but it can't do anything because it does not have any source. We must create a source object from one of the possible sources available in `bge.texture`:

`VideoFFmpeg`
: Moving pictures. Video file, video capture, video streaming.

`ImageFFmpeg`
: Still pictures. Image file, image on web.

`ImageBuff`
: Image from application memory. For computer generated images, drawing applications.

`ImageViewport`
: Part or whole of the viewport (=rendering of the active camera displayed on screen).

`ImageRender`
: Render of a non active camera.

`ImageMix`
: A mix of 2 or more of the above sources.

In this example we use a simple video file as source. The `VideoFFmpeg` constructor takes a file name as argument. To avoid any confusion with the location of the file, we will use `GameLogic.expandPath ()` to build an absolute file name, assuming the video file is in the same directory as the blend file:

```
movie = GameLogic.expandPath('//trailer_400p.ogg')
GameLogic.video.source = bge.texture.VideoFFmpeg(movie)
```

We create the video source object and assign it to the `Texture` object `source` attribute to set the source and make it persistent: as the `Texture` object is persistent, the source object will also be persistent.

Note that we can change the `Texture` source at any time. Suppose we want to switch between two movies during the game. We can do the following:

```
GameLogic.mySources[0] = bge.texture.VideoFFmpeg('movie1.avi')
GameLogic.mySources[1] = bge.texture.VideoFFmpeg('movie2.avi')
```

And then assign (and reassign) the source during the game:

```
GameLogic.video.source = GameLogic.mySources[movieSel]
```

Setup the source

The `VideoFFmpeg` source has several attributes to control the movie playback:

`range`
: [start,stop] (*floats*). Set the start and stop time of the video playback, expressed in seconds from beginning. By default the entire video.

`repeat`
: (*integer*). Number of video replay, -1 for infinite.

`framerate`
: (*float*). Relative frame rate, <1.0 for slow, >1.0 for fast.

`scale`
: (*bool*). Set to True to activate fast nearest neighbor scaling algorithm. Texture width and height must be a power of 2. If the video picture size is not a power of 2, rescaling is required. By default `bge.texture` uses the precise but slow `gluScaleImage()` function. Best is to rescale the video offline so that no scaling is necessary at runtime!

`flip`
: (*bool*). Set to True if the image must be vertically flipped. FFmpeg always delivers the image upside down, so this attribute is set to True by default.

`filter`
: Set additional filter on the video before sending to GPU. Assign to one of `bge.texture` filter object. By default the image is send unchanged to the GPU. If an alpha channel is present in the video, it is automatically loaded and sent to the GPU as well.

We will simply set the `scale` attribute to True because the `gluScaleImage ()` is really too slow for real time video. In case the video dimensions are already a power of 2, it has no effect.

```
GameLogic.video.source.scale = True
```

Play the video

We are now ready to play the video:

```
GameLogic.video.source.play()
```

Video playback is not a background process: it happens only when we refresh the texture. So we must have another script that runs on every frame and calls the `refresh ()` method of the `Texture` object:

```
if hasattr(GameLogic, 'video'):
GameLogic.video.refresh(True)
```

If the video source is stopped, `refresh()` has no effect. The argument of `refresh ()` is a flag that indicates if the texture should be recalculated on next refresh. For video playback, you definitively want to set it to True.

Checking video status

Video source classes (such as VideoFFMpeg) have an attribute `status`. If video is playing, its value is 2, if it's stopped, it's 3. So in our example:

```
if GameLogic.video.source.status == 3:
#video has stopped
```

Advanced work flow

True argument in `Texture.refresh()` method simply invalidates the image buffer after sending it to the GPU so that on next frame, a new image will be loaded from the source. It has the side effect of making the image unavailable to Python. You can also do it manually by calling the `refresh ()` method of the source directly.

Here are some possible advanced work flow:

- Use the image buffer in python (doesn't effect the Texture):

```
GameLogic.video.refresh(False)
image = GameLogic.video.source.image
# image is a binary string buffer of row major RGBA pixels
# ... use image
# invalidates it for next frame
GameLogic.video.source.refresh()
```

- Load image from source for Python processing without download to GPU:
- note that we don't even call refresh on the Texture
- we could also just create a source object without a Texture object

```
image = GameLogic.video.source.image
# ... use image
GameLogic.video.source.refresh()
```

- If you have more than 1 material on the mesh and you want to modify a texture of one particular material, get its ID

```
matID = bge.texture.materialID(gameobj, "MAmat.001")
```

GLSL material can have more than 1 texture channel, identify the texture by the texture slot where it is defined, here 2

```
tex=bge.texture.Texture(gameobj, matID, 2)
```

Advanced demos

Here is a demo that demonstrates the use of two videos alternatively on the same texture. Note that it requires an additional video file which is the elephant dream teaser. You can replace with another other file that you want to run the demo.

Here is a demo that demonstrates the use of the `ImageMix` source. `ImageMix` is a source that needs sources, which can be any other `Texture` source, like `VideoFFmpeg`, `ImageFFmpeg` or `ImageRender`. You set them with `setSource ()` and their relative weight with `setWeight()`. Pay attention that the weight is a short number between 0 and 255, and that the sum of all weights should be 255. `ImageMix` makes a mix of all the sources according to their weights. The sources must all have the same image size (after reduction to the nearest power of 2 dimension). If they don’t, you get a Python error on the console.