BGECore

Introduction

What's BGECore?

You can think of BGECore as a framework for Blender Game Engine. BGECore provides you with the tools to make great games in BGE. It adds functionality to BGE through Python and makes coding typically complex setups easy.

Why a framework and not a library?

The lack of a standard in BGE makes impossible writing a library that could work with any design a developer could think of. That's why in order to provide new functionality BGECore requieres of a strong standard that every BGECore user must follow. This may seem like a restriction to you now, but the more you use BGECore the more you'll realize how much important is to follow the BGECore Design. The BGECore design will not bound your skills, it will mantain your sanity.

The most basic rules

1. No logic bricks or game properties are allowed, all the game is done in Python.

2. Your project must follow the standard BGECore directory organization, including the launcher.

3. There must be one and only one GUI scene, all scenes can have only one scene Behavior.

License

Table of contents

Directory organization

All filenames or dirnames are lowercase separated by underscores and singular. The following is an example of a blender game organization design. **Bold** are mandatory for any game and are explained in more detail in the following pages.

• **engine/**	Here we put blenderplayer and all it's required files (*.dll, python, etc...)
• **data/**	Blend files, python packages, and other miscellaneous light files.
◦ **core/**	Python libraries for BGE, used from scripts. Don't put here generic Python libraries.
◦ **script/**	Scripts for a specific game. Your game is basically coded here.
▪ **__init__.py**	This file is used by python to make a package.
▪ **constant.py**	It contains constants that you will use all across your game.
▪ **gui.py**	The GUI scene behavior and its elements are found here.
▪ **behavior/**	All behaviors not being part of the GUI are here.
▪ **control.py**	This file should contain an API accessible from any point in the game.
◦ **gui/**	We put here textures and files related to the GUI
◦ **quest/**	Almost all games have rules, they can be coded here.
◦ dialog/	This folder is not part of BGECore but may be useful.
▪ intro.xml	We could have a dialog system based on xml files.
◦ **game.blend**	Your main blend file.
• **config.txt**	This file contains generic configurations of your game.
• **MyGame.exe**	The game launcher. It's different for Linux and MacOS.
• readme.txt	A readme file is always welcome.

The“constant.py” file

Constants in BGECore are defined using UPPERCASE. If you see a variable witted like that you know it is a constant and you shouldn't modify it. A basic constant.py file should look like:

```
VERSION = “0.2”

CORE_SCENE_GUI = “GUI”
CORE_DEBUG_PRINT = True
CORE_DEBUG_VERBOSE = True
```

Constants prefixed with “CORE” are used from inside BGECore and therefore should not be deleted. In this case `CORE_SCENE_GUI` is used to know the name of the GUI scene, whilst `CORE_DEBUG_PRINT` and `CORE_DEBUG_VERBOSE` are used by the functions `debug()` and `verbose()` from `core.utils`. BGECore uses those to display warnings when something is not setup correctly but the game can still continue. You can also use them in your game.

The main blend file

The main blend file needs to have a minimum setup to work with BGECore, since it is not be trivial a template file ready to use is provided.

With the template file, as long as you read this guide and don't delete anything substantial, there shouldn't be any problems, however for better understanding here is a list of the things that your main blend file must have.

- A scene for the GUI which name coincides with `constant.CORE_SCENE_GUI` is mandatory.
- All names must have CamelCase starting with a capital letter. Acronyms, if used as a prefix, can be witted in uppercase, e.j. GUICamera or GUIScreen.
- Objects that will potentially be duplicated must be enumerated in the standard blender way, e.j. GUICamera.000.
- Font objects must be preceded with a "Font." suffix and be placed in a hidden layer of the GUI scene, more information in the Labels section.
- Cameras in the GUI scene must be orthographic, have a rotation of (0,0,0) and a position on the Z axis of 10. It's also recommended to have an orthographic scale of 16.
- The main camera of the GUI scene "GUICamera.000" or "GUICamera" must have the following logic bricks:
    - `ALWAYS (pulse_mode = True) → PYTHON (module = "main.loop")`
- All other scenes should have the following logic brick in their main camera:
    - `ALWAYS → PYTHON(module = "main.autoStart")`
- No more logic bricks are allowed in any scene.
- There must be a GUIScreen object in a hidden layer of the GUI scene.
- A main.py file should be packed inside the blend, it must not be modified. The content of that file is as follows:

```
import core, script

def loop():
    core.interface.loop()
    core.game.loop()

def autoStart():
    from bge import logic
    if not core.utils.getSceneByName(script.constant.CORE_SCENE_GUI):
        sc = logic.getCurrentScene()
        sc.replace(script.constant.CORE_SCENE_GUI)
```

Launching and distributing your game

BGECore comes with the “engine” folder empty. That's because its contents will change depending on which operating system and blender version you are using or you want to publish your game to. For example if you want to publish your game for Windows users you will put a copy of Blender for windows in there. For other operating systems you can just download their version of Blender and put that in the engine folder.

However, now you have a folder containing all Blender inside, it should be around 200mb which is a lot for just the game engine. If you don't want to distribute Blender together with your game you can just delete the parts of it that the game will not be using. The following scheme shows in **bold** the parts that are required for BGE, in *cursive* those parts that may or may not be needed and in default those who can be deleted. This scheme is for Windows, other OS can difeer.

- **2.76/**
    - datafiles/
    - **python/**
        - bin/
        - **lib/**
    - scripts/
- blender.exe
- blender-app.exe
- **blenderplayer.exe**
- **.txt*
- ***.dll**

TODO: Standard Launcher

Since now your game is well organized in folders and no resources are external to the game folder, your are ready for distributing your game. Most common ways of distributing your game is in a compressed file (“zip”, “rar”, “7z”) or with an installer. Notice that installers only work on Windows. To make an installer you can use applications like the ones in this link: http://www.techsupportalert.com/content/best-free-setup-builder.htm

In Windows you also have to possibility to change the icon of your game (Linux and MacOS don't use custom icons on executable files), for that, have a look at Resource Hacker^{TM}

Note: Blenderplayer and Blender are being merged recently, this makes newer versions of BGE bigger. e.j. `BGE 2.56 Win64 → 55mb` vs. `BGE 2.76 Win64 → 83mb`

The Graphical User Interface (GUI)

It has never been easy to make GUIs in Blender Game Engine. Such is the difficulty that some libraries based on the openGL module had appeared. Those libraries are useful to make GUIs but they are hard to use and they don't make use of blender at all. That's why the standard interface module of BGECore doesn’t work like that. The interface module doesn't crate GUIs from scratch, instead, it uses game objects as the base for all kind of widgets and gui elements.

The GUI scene behavior

For general documentation about behaviors go to the Behaviors section.

The GUI scene has one and only one scene behavior. This behavior and any other elements related to the GUI should go in the gui.py file. If the gui.py file becomes too big, it can be converted into a package. In the initialization of this behavior we declare the interface elements, e.j.

```
from core import behavior
from script import constant

class GUI(behavior.Scene):
	def init(self):
		#do something here… e.j.
		self.wid1 = MyWidget(“Sprite.000”)
		self.wid2 = MyWidget(“Sprite.001”)

behavior.addScene(constant.CORE_SCENE_GUI, GUI)
```

Widgets

The widget is the most basic kind of element, almost all other events are derived from it. A widget is a basic object that can receive mouse events and react accordingly in return. The following is a simple example of how to use a basic widget element in a game. This example should be combined with the first one.

```
from core.interface.widget import Widget
from core import utils

class MyWidget(Widget):
	def mouse_click(self):
		utils.verbose(“Hello World”)
```

With this, clicking “Sprite.000” or “Sprite.001” would print on the terminal “Hello World”. Widgets also have some other useful methods that you will also find on other GUI elements:

`Widget(obj)`	Constructor.
`delete()`	Deletes the widget but doesn't delete the associate game object.
`self.obj`	The associated object, e.j. “Sprite.000”.
`mouse_in()`	Event.
`mouse_out()`	Event.
`mouse_over()`	Not implemented yet.
`mouse_click()`	Event.

`self.transformable.append()`	Adds a game object to the transformable list.
`self.transformable.remove()`	Removes a game object from the transformable list.
`getPosition()`	Return the associated object position.
`setPosition(x, y, z)`	Sets the widget position moving all objects in the transformable list. If any argument is None that axis remains unchanged.
`getScale()`	…
`setScale()`	…
`getRotation()`	In radians.
`setRotation()`	In radians.

The window

We have seen how handle mouse events, but how are they generated in the first place? They are generated in the window object. The window object (ref `core.interface.window`) is a singleton of the class Window. This object generates mouse events using a rayCast technique like the mouse sensor does with logic bricks. The window also controls the cursor.

`Window()`	Constructor.
`resize(x, y)`	Changes the size of the window conserving the aspect ratio.
`hideCursor()`	Hides the cursor.
`showCursor()`	Shows the cursor.
`setCursor(objName)`	Changes the cursor for a object in a hidden layer of the GUI scene. If `objName` is `None` it restores the default cursor.

Note: Current version only supports orthographic cameras, events generated in another configuration could not work or be bugged.

Buttons

A button works just like a widget but it can spawn an overlaying game object when the mouse is over it (like a real button does). Shortcut: `core.interface.Button`

`[Widget] Button(obj, spawn)`	Constructor where `spawn` is the name of an object on a GUI inactive layer.
`enable()`	Enables the button
`disable()`	Disables the button. Stops events and hides game objects.

You also have the TextButton variant in which you can use to have the same Button displaying different Labels on them, very useful for menus.

`[Button] TextButton(obj, spawn, font, text, size, align)`	Constructor, default: `size=16, align=ALIGN_LEFT`
`self.text`	The Label object.

Notice that objects that shouldn't be detected by the window raycast must be set to **no collision**. This includes fonts, cursors, the screen, and sprites that go over a button.

Labels, presentable text in BGE

This is one of the things that has been more difficult to do in BGE since immemorial times. Now with BGECore it's easy. BGECore uses the classic dynamic text objects (form the days of 2.49) as font objects. They are spawned to the GUI scene creating real time texts. This is how we spawn a Label in BGECore

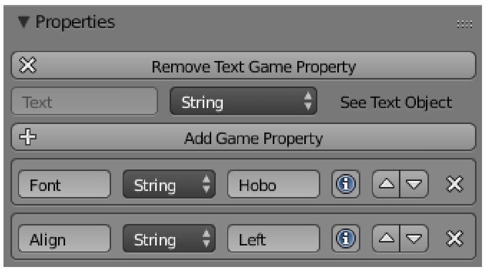
```
from core.interface import Label
       …

       Label("Hobo", "Hello World!", 16, interface.label.ALIGN_CENTER, [0,0,0])
```

This will spawn the text "Hello World!" with the font "Hobo" and a size of 16 at the center of the screen. Labels have the following properties:

`[Widget] Label(font, text, size, align, position)`	Constructor, where by default: `size=16, align=ALIGN_LEFT, position=[0,0,0]`
`getText()`	Returns a string matching the text being displayed.
`setText()`	Changes the text being displayed.
`getFont()`	Returns the dynamic text object representing the font.
`setFont(font)`	Changes the font used to display the text, where `font` is a string.
`getSize()`	Returns the size of the text.
`setSize(size)`	Changes the size of the text, called by the constructor.
`getAlign()`	Returns a number representing the alignment of the text. Can be compared with `ALIGN_LEFT`, `ALIGN_CENTER` or `ALIGN_RIGHT`
`setAlign(align)`	Changes the alignment.
`visible(bool)`	Default: `bool=None`, if `bool=True` makes the label visible, if `bool=False` makes the label invisible, if `bool=None` returns the visibility state.

However, if we can only spawn texts we lost the ability to create texts in the viewport. The text object of blender, thought it works very poorly in the BGE, has its advantages: You can move it around to see how it looks and you can change it "in-place". For this reason BGECore also makes use of the new text objects, replacing them with Labels at the start of the game. Notice that GUI elements should always be on the GUI scene, therefore text objects in other scenes will not be replaced.



Font | String | Hobo
Align | String | Left

This image shows the properties that a text object must have in order to be replaced correctly. If any of this properties is missing the replacement will fail and a debug message will be print. Notice that this properties will not modify the original text object, to have a correct representation you should update the text settings.

A Label created from a text object can be accessed with `core.module.labels["ObjectName"]`

Creating custom fonts

The font objects used by Labels on BGECore must follow some rules:

- They must be in a hidden/inactive layer of the GUI scene.
- Their name must be prefixed with “Font.”, e.j. “Font.Hobo”
- The mesh, the material, and the texture must have the same name than the object.
- They must have a “Text” property that represents the displaying text.
- The material must have “Backface Culling” disabled.

To create a new font you can just duplicate the “Font.Hobo” object and change its texture to the texture of your font. For a better explanation of how Dynamic Texts are created in BGE check this video: https://www.youtube.com/watch?v=GKO4jfAHtjk

Sometimes, after a modification, a font object stops working and all texts display “@@@...”, in such cases restarting blender usually fixes the problem.

Some useful tools, not part of BGECore, are also provided with it. One of these is JBFT, an application that will help you to create font textures from font files. Path: “tools/Font Generator/jbft.jar”. It's recommended to create textures with a resolution of 1024×1024.

A font texture can also be modified to create some cool effects, check this tutorial: http://www.blendenzo.com/tutColoredText.html

Note: Alignment of Labels are done using `bge.logic.NewLib` which can be bugged in some versions of Blender prior to 2.76

Menus

Buttons and TextButtons have a problem, each new item needs a new class to work. Menus allow Buttons and TextButtons to work in the same class using a state machine. e.j:

```
from core import interface, behavior

class GUI(behavior.Scene):
    def init(self):
        MyMenu(0, “Sprite.000”, “Sprite.001”, “Hobo”, “Button0”)
        MyMenu(1, “Sprite.000”, “Sprite.001”, “Hobo”, “Button1”)

MyMenu(interface.TextMenu):
    def mouse_click(self):
        if self.index == 0:
            print(“I'm the button 0”)
         if self.index == 1:
            print(“I'm the button 1”)
```

TO DO: Spawnable menus.

Behaviors

We have two types of Behaviors, scene behaviors and object behaviors.

```
core.behavior.Object
core.behavior.Scene
```

Each scene can have at most only one Behavior. BGECore uses always only two scenes running at the same time, hence two behaviors. The GUI scene remains always the same, however you can change the game scene as much as you want. To change the game scene you can't use BGE methods, instead use `utils.setScene(“SceneName”)`.

You can have access to the running scenes and its behaviors at any time using the module namespace, e.j:

```
from core import module

game_scene = module.scene_game
gui_scene = module.scene_gui
game_scene_behavior = module.scene_behavior
gui_scene_behavior = module.scene_gui_behavior
```

Behaviors have two main methods, `init` and `update`. Init is called when the behavior is loaded while update is called every frame. When initializing a scene behavior you should define other behaviors of that scene and any variables you'll be using in that scene. A basic scene behavior looks like this:

```
from core import behavior
from core import module
from script.behavior.camera import CameraLock

class MySceneBehavior(behavior.Scene):
	def init(self):
		#initialize other behaviors or variables here, e.j:
		self.camlock = CameraLook(module.scene_game.active_camera)

	def update(self):
		pass

behavior.addScene(“SceneName”, MySceneBehavior)
```

The `addScene` function will add our game scene to a dictionary (`module.scene_behavior_dict`). When you set the game scene a new instance of that behavior will be crated and the old one will be deleted. It's common to use the same scene behavior in all game scenes if our game scenes are actually levels but all them work the same. In this case we can use static variables in our class and this way we don't lost data between scenes. E.j:

```
from core import behavior, module, utils

class MySceneBehavior(behavior.Scene):
	coins = None

	def init(self):
		if not self.coins: self.coins = 0

	def update(self):
		self.coins += 1
		if self.coins == 1000:
			utils.setScene(“SecondScene”)
		if self.coins == 2700:
			utils.setScene(“ThirdScene”)

behavior.addScene(“FirstScene”, MySceneBehavior)
behavior.addScene(“SecondScene”, MySceneBehavior)
```

```
It will continue…
```

Quests and inventories

Todo

Saving your game

Todo

The utils module

Todo

The media module

Todo