demosys-py Documentation

Release 0.3.5

Einar Forselv

Apr 22, 2017

Contents:

CHAPTER 1

demosys-py

A python 3 implementation of a C++ project used to create and prototype demos (see demoscene) in OpenGL. The design of this version is heavily inspired by the Django project.

This is a cross platform project that should work on OS X, Linux and Windows.

This was originally made for for non-interactive real time graphics combined with music ("real time music videos"). It's made for people who enjoy playing around with modern OpenGL without having to spend lots of time creating all the tooling to get things up and running.

Getting Started

Python 3

Make sure you have Python 3 installed. On Windows and OS X you can simply install the latest Python 3 by downloading an installer from the official Python site.

Note: We recommend Python 3.6 or higher because of general **speed improvements** of the language, but Python versions down to 3.4 should work.

Most linux distributions already have at least Python 3.4 installed thought `python3`. See documentation for your distribution on how to install a newer versions.

It is common to have multiple versions of Python installed on all operating systems.

Binary Dependencies

We use glfw for creating an OpenGL context, windows and handling keyboard and mouse events. This is done thought the pyGLFW package that is a `ctype` wrapper over the origonal GLFW api. You will have to install the actual library yourself.

We require glfw 3.2.1 or later.

OS X

> brew install gflw

Linux

glfw should be present in the vast majority of the package managers.

Windows

Download binaries from the glfw website.

We do also support audio playback that will need additional dependencies, but this is covered in a different section.

Create a virualenv

First of all create a directory for your project and navigate to it using a terminal. We assume Python 3.6 here.

OS X / Linux

```
python3.6 -m pip install virtualenv
python3.6 -m virtualenv env
source env/bin/activate
```

Windows

```
python36.exe -m pip install virtualenv
python36.exe -m virtualenv env
\env\Scripts\activate
```

We have now created an isolated Python environment and are ready to install packages without affecting the Python version in our operating system.

Setting up a Project

First of all, install the package (you should already be inside a virtualenv)

> pip install demosys-py

The package will add a new command `demosys-admin` we will now use to create a project.

> demosys-admin createproject myproject

This will generate the following files:

```
myproject
- settings.py
manage.py
```

- `settings.py` is the settings for your project
- `manage.py` is an executable script for running your project

In order to get something to the screen we have to make an effect.

```
cd myproject
demosys-admin createeffect cube
```

We should now have the folloing structure:

```
myproject/
- cube
|    - effect.py
|    - shaders
|    |    - cube
|    |        - default.glsl
|    - textures
|        - cube
- settings.py
manage.py
```

The `cube` directory is a template for an effect: - The standard `effect.py` module containing a single `Effect` implementation - A local `shaders` directory for glsl shaders specific to the effect - A local `textures` directory for texture files specific to the effect

Notice that the `shaders` and `textures` directory also has a sub-folder with the same name as the effect. This is because these directories are added to a global virtual directory and the only way to make these resources unique is to put it in a directory that is *hopefully* unique.

This can of course be used in creative ways to also override resources on purpose.

For the effect to be recognized by the system we need to add it `EFFECTS` in `settings.py`.

```
EFFECTS = (
    'myproject.cube',  # Remember comma!
)
```

As you can see, effects are added by using the python package path. Where you put effect packages is entirely up to you, but a safe start is to put them inside the project package as this removes any possibility of effect package names colliding with other python packages.

We can now run the effect that shows a spinning cube!

```
./manage.py runeffect myproject.cube
```

Controls

Basic Keyboard Controls

- `ESC` to exit
- `SPACE` to pause the current time
- `X` for taking a screenshot (see settings)

Camera Controls

You can include a system camera in your effects through `self.sys_camera`. Simply apply the `view_matrix` of the camera to your transformations.

Keyboard Controls:

- `W` forward
- `S` backwards

- `A` strafe left
- `D` strafe right
- `Q` down the y axis
- `E` up the y axis

Mouse Controls

- Standard yaw/pitch camera rotation with mouse

Settings

The `settings.py` file must be present in your project and contains (you guessed right!) settings for the framework. This is pretty much identical to Django.

OPENGL

Using these values you are sure it will run on all platforms. OS X only support forward compatible core contexts. This will bump you to the latest version you drivers support.

```
OPENGL = {
    "version": (4, 1),
    "profile": "core",
    "forward_compat": True,
}
```

The default opengl version is 4.1. Some older systems might need that tuned down to 3.3, but generally 4.1 is widely supported.

WINDOW

Window properties. If you are using Retina display, be aware that these values refer to the virual size. The actual buffer size will be 2 x.

```
WINDOW = {
    "size": (1280, 768),
    "vsync": True,
    "resizable": False,
    "fullscreen": False,
    "title": "demosys-py",
    "cursor": False,
}
```

MUSIC

If `MUSIC` is defined, demosys will attempt to play. (We have only tried mp3 files!)

Note: Getting audio to work requires additional setup.

```
PROJECT_DIR = os.path.dirname(os.path.abspath(__file__))
MUSIC = os.path.join(PROJECT_DIR, 'resources/music/tg2035.mp3')
```

TIMER

This is the timer class that controls time in your project. This defaults to `demosys.timers.Timer` that is simply keeps track of system time using `glfw`.

```
TIMER = 'demosys.timers.Timer'
```

Other timers are:

- `demosys.timers.MusicTimer` requires `MUSIC` to be defined and will use the current time in an mp3.
- `demosys.timers.RocketTimer` is the same as the default timer, but uses uses the rocket library.
- `demosys.timers.RocketMusicTimer` requires `MUSIC` and `ROCKET` to be configured.

ROCKET

Configuration of the pyrocket sync-tracker library.

- `rps`: Number of rows per second
- `mode`: The mode to run the rocket client - `editor`: Requires a rocket editor to run so the library can connect to it - `project`: Loads the project file created by the editor and plays it back - `files`: Loads the binary track files genrated by the client through remote export in the editor.
- `project_file`: The absolute path to the project file
- `files`: The absolute path to the directory containing binary track data

```
ROCKET = {
    "rps": 24,
    "mode": "editor",
    "files": None,
    "project_file": None,
}
```

EFFECTS

Effect packages demosys will initialize and use (Same as apps in Django).

```
EFFECTS = (
    'myproject.cube',
)
```

SHADER_DIRS/FINDERS

`DIRS` contains absolute paths the `FileSystemFinder` will look for shader while `EffectDirectoriesFinder` will look for shaders in all registered effects in the order they were added.

The `FileSystemFinder` will look in all paths specified in `SHADER_DIRS`. All paths must be absolute (just join on `PROJECT_DIR`). This is a good way to add project-global shaders used by multiple effecst.

```
SHADER_DIRS = (
    os.path.join(PROJECT_DIR, 'resources/shaders'),
)

SHADER_FINDERS = (
    'demosys.core.shaderfiles.finders.FileSystemFinder',
    'demosys.core.shaderfiles.finders.EffectDirectoriesFinder',
)
```

TEXTURE_DIRS/FINDERS

Same principle as `SHADER_DIRS` and `SHADER_FINDERS`.

```
# Hardcoded paths to shader dirs
TEXTURE_DIRS = (
    os.path.join(PROJECT_DIR, 'resource/textures'),
)

# Finder classes
TEXTURE_FINDERS = (
    'demosys.core.texturefiles.finders.FileSystemFinder',
    'demosys.core.texturefiles.finders.EffectDirectoriesFinder'
)
```

SCREENSHOT_PATH

Absolute path to the directory screenshots will be saved. If not defined or the directory don't exist, the current working directory will be used.

```
SCREENSHOT_PATH = os.path.join(PROJECT_DIR, 'screenshots')
```

Temporary Notes

This needs to be restructured into actual docs.

- Shaders and textures can be easily loaded by using the `get_texture` and `get_shader` method inherited from `Effect`.
- The `cube` objects is a `VAO` that you bind supplying the shader and the system will figure out the attribute mapping.
- Please look in the `demosys.opengl.geometry` module for the valid attribute names and look at shaders in the testdemo.
- You currently define vertex, fragment and geometry shader in one glsl file separated by preprocessors.
- Effects not defined in the `settings.py` module will not run.

That should give you an idea..

Anything we draw to the screen must be implemented as an `Effect`. If that effect is one or multiple things is entirely up to you. An effect is an individual package/directory containing an `effect.py` module. This package can also contain a `shaders` and `textures` directory that demosys will automatically find and load resources from. See the testdemo.

Explore the testdemo project, and you'll get the point.

Some babble about the current state of the project:

- All geometry must be defined using VAOs. There's a very convenient VAO class for this already making it quick and easy to create them. Look at the `demosys.opengl.geometry` module for examples.
- We support vertex, fragment and geometry shaders for now. A program must currently be written in one single `.glsl` file separating the shaders with preprocessors. See existing shaders in testdemo.
- The Shader class will inspect the linked shader and cache all attributes and uniforms in local dictionaries. This means all `uniform*`-setters use the name of the uniform instead of the location. Location is resolved internally in the object/class.
- The VAOs `bind(..)` requires you to pass in a shader. This is because the VAO will automatically adapt to the attributes in your shader. During the VAO creation you need to make the name mapping to the attribute name. If you have a VAO with positions, normals, uvs and tangents and pass in a shader that only use position (or any other combination of attributes in the VAO); the VAO class will on-the-fly generate a version internally with only positions.
- We only support 2D textures at the moment loaded with PIL/Pillow, but this is trivial to extend.
- Resource loading is supported in the `Effect` class itself. In `__init__()` you can fetch resources using for example `self.get_shader` or`self.get_texture`. This will return a lazy object that will be populated after the loading stage is done.
- Resources shared between effects can be put outside effect packages inside your project directory. For example in `testdemo/resources/shaders` and `testdemo/resources/textures`. Make sure you add those paths in the settings file.
- We don't have any scene/mesh loaders. You can hack something in yourself for now or just stick to or extend the `geometry` module. - We try to do as much validation as possible and give useful feedback when something goes wrong.
- The `time` value passed to the effects `draw` method is the current duration in the playing music. If no music is loaded, a dummy timer is used.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search