demosys-py Documentation

Release 0.3.6

Einar Forselv

Jul 04, 2018

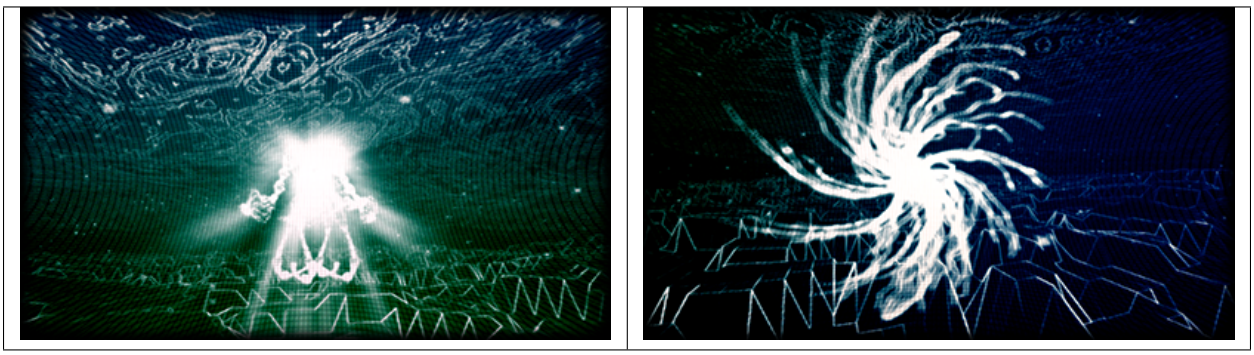
Contents:

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This was originally made for for non-interactive real time graphics combined with music ("real time music videos").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.

1.1 Getting Started

1.1.1 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.

1.1.2 Binary Dependencies

We use glfw for creating an OpenGL context, windows and handling keyboard and mouse events. This is done though the pyGLFW package that is wrapper over the original glfw library. You will have to install the actual library yourself.

We require glfw 3.2.1 or later.

OS X

```
brew install glfw
```

Linux

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

Windows

Download binaries from the glfw website. You can drop the dll in the root of your project.

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

1.1.3 Create a virtualenv

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 versions in our operating system.

1.1.4 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 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

More information about projects can be found in the *Project* section.

1.1.5 Creating an Effect

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

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

We should now have the following 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 them in a directory.

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 to `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
```

1.2 Controls

1.2.1 Basic Keyboard Controls

- `ESC` to exit
- `SPACE` to pause the current time (tells the configured timer to pause)

- `X` for taking a screenshot (output path is configurable in settings)

1.2.2 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
- `R` reload all shaders

Mouse Controls

- Standard yaw/pitch camera rotation with mouse

1.3 Settings

The `settings.py` file must be present in your project containing settings for the project.

When running your project with `manage.py`, the script will set the `DEMOSYS_SETTINGS_MODULE` environment variable. This tells the framework where it can import the project settings. If the environment variable is not set the project cannot start.

1.3.1 OPENGL

> **Warning:** We cannot guarantee the framework will work properly for non-default values. We strongly discourage enabling backwards compatibility. It might of curse make sense in some cases if you bring in existing draw code from older projects. Be extra careful when using deprecated OpenGL states.

Using default values we can be more confident that cross-platform support is upheld. Remember that some platforms/drivers such as on OS X, core profiles can only be forward compatible or the context creation will simply fail.

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.

```
OPENGL = {
    "version": (4, 1),  # 3.3 -> 4.1 is acceptable
    "profile": "core",
    "forward_compat": True,
}
```

- `version` describes the major and minor version of the OpenGL context we are creating

- `profile` should ideally always be `core`, but we leave it configurable for those who might want to include legacy OpenGL code permanently or temporary. Do note that not using core profile will exclude the project from working on certain setups and may have unexpected side effects.
  - `any`: glfw.OPENGL_ANY_PROFILE,
  - `core`: glfw.OPENGL_CORE_PROFILE,
  - `compat`: glfw.OPENGL_COMPAT_PROFILE,
- `forward_compat` True, is required for the project to work on OS X and drivers only supporting forward compatibility.

Note: To make your project work on OS X you cannot move past version 4.1 (sadly). This doesn't mean we cannot move past 4.1, but as of right now we focus on implementing features up to 4.1.

1.3.2 WINDOW

Window/screen properties. If you are using Retina or 4k displays, be aware that these values can refer to the virtual size. The actual buffer size will be larger.

```
WINDOW = {
    "size": (1280, 768),
    "aspect_ratio": 16 / 9,
    "fullscreen": False,
    "resizable": False,
    "vsync": True,
    "title": "demosys-py",
    "cursor": False,
}
```

- `size`: The window size to open. Note that on 4k displays and retina the actual frame buffer size will normally be twice as large. Internally we query glfw for the actual buffer size so the viewport can be correctly applied.
- `aspect_ratio` is the enforced aspect ratio of the viewport.
- `fullscreen`: True if you want to create a context in fullscreen mode
- `resizable`: If the window should be resizable. This only applies in windowed mode. Currently we constrain the window size to the aspect ratio of the resolution (needs improvement)
- `vsync`: Only render one frame per screen refresh
- `title`: The visible title on the window in windowed mode
- `cursor`: Should the mouse cursor be visible on the screen? Disabling this is also useful in windowed mode when controlling the camera on some platforms as moving the mouse outside the window can cause issues.

The created window frame buffer will by default use:

- RGBA8 (32 bit per pixel)
- 32 bit depth buffer were 24 bits is for depth and 8 bits for stencil
- Double buffering
- color, depth and stencil is cleared every frame

1.3.3 MUSIC

The `MUSIC` attribute is used by timers supporting audio playback. When using a timer not requiring an audio file, the value is ignored. Should contain a string with the absolute path to the audio file.

Note: Getting audio to work requires additional setup. See the *Audio* section.

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

1.3.4 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.

More information can be found in the *Timers* section.

1.3.5 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,
}
```

1.3.6 EFFECTS

Effect packages that will be recognized by the project. Initialization should happens in the order they appear in the list.

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

1.3.7 EFFECT_MANAGER

Effect mangers are pluggable classed that initialize and run effects. When only having a single effect we can run it using `runeffect`, but when having multiple effects we need something to decide what effect should be active.

The default effect manager is the `SingleEffectManager` that is also enforced when running `./manage.py runeffect <name>`. If we use the `run` sub-command, the first registered effect will run.

```
EFFECT_MANAGER = 'demosys.effects.managers.single.SingleEffectManager'
```

More info in the *Effect Managers* section.

1.3.8 SHADER_STRICT_VALIDATION

Boolean value. If `True` shaders will raise `ShaderError` when for example setting uniforms variables that don't exist or is of the wrong type.

If the value is `False` an error message will be generated instead.

This is useful when working with shaders. Sometimes you want to allow missing or incorrect uniforms. Other times you want to know in a more brutal way that something is wrong.

1.3.9 SHADER_DIRS/FINDERS

`SHADER_DIRS` contains absolute paths the `FileSystemFinder` will look for shaders.

`EffectDirectoriesFinder` will look for shaders in all registered effects in the order they were added. This assumes you have a `shaders` directory in your effect package.

```
# Register a project-global shader directory
SHADER_DIRS = (
    os.path.join(PROJECT_DIR, 'resources/shaders'),
)

# This is the defaults is the property is not defined
SHADER_FINDERS = (
    'demosys.core.shaderfiles.finders.FileSystemFinder',
    'demosys.core.shaderfiles.finders.EffectDirectoriesFinder',
)
```

1.3.10 TEXTURE_DIRS/FINDERS

Same principle as `SHADER_DIRS` and `SHADER_FINDERS`.

```
# Absolute path to a project-global texture directory
TEXTURE_DIRS = (
    os.path.join(PROJECT_DIR, 'resources/textures'),
)
```

(continues on next page)

(continued from previous page)

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

1.3.11 SCENE_DIRS/FINDERS

Same principle as `SHADER_DIRS` and `SHADER_FINDERS`. This is where scene files such as wavefront and gltf files are loaded from.

```
# Absolute path to a project-global scene directory
SCENE_DIRS = (
    os.path.join(PROJECT_DIR, 'resources/scenes'),
)

# Finder classes
SCENE_FINDERS = (
    'demosys.core.scenefiles.finders.FileSystemFinder',
    'demosys.core.scenefiles.finders.EffectDirectoriesFinder'
)
```

1.3.12 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')
```

1.4 Project

Before we can do anything with the framework we need to create a project. A project is simply a package containing a `settings.py` module. In addition you need an entrypoint script.

This can be auto-generated using the `demosys-admin` command:

```
demosys-admin createproject myproject
```

This will generate the following structure:

```
myproject
└── settings.py
manage.py
```

- `settings.py` is the settings for your project with good defaults. See *Settings* for more info.
- `manage.py` is an executable script for running your project

1.4.1 Effects

It's normally a good idea to put effect packages inside the project package as this protects you from package name collisions. It's of course also fine to put them at the same level as your project or even have them in separate repositories and install them as packages thought `pip`.

1.4.2 manage.py

The `manage.py` script is an alternative entry point to `demosys-admin`. Both can perform the same commands. The main purpose of `demosys-admin` is to initially have an entry point to the commands creating a projects and effects when we don't have a `manage.py` yet.

1.4.3 Management Commands

Custom commands can be added to your project. This can be useful when you need additional tooling or whatever you could imagine would be useful to run from `manage.py`.

Creating a new command is fairly straight forward. Inside your project package, create the `management/commands/` directories. Inside the commands directory we can add commands. Let's add the command `test`.

The project structure (excluding effects) would look something like:

```
myproject
└── settings.py
└── management
    └── commands
        └── test.py
```

Notice we added a `test` module inside `commands`. The name of the module will be name of the command. We can reach it by:

```
./manage.py test
```

Our test command would look like this:

```
from demosys.core.management.base import BaseCommand

class Command(BaseCommand):
    help = "Test command"

    def add_arguments(self, parser):
        parser.add_argument("message", help="A message")

    def handle(self, *args, **options):
        print("The message was:", options['message'])
```

- `add_arguments` exposes a standard argparser we can add arguments for the command.
- `handle` is the actual command logic were the parsed arguments are passed in
- If the parameters to the command do not meet the requirements for the parser, a standard arparse help will be printed to the terminal
- The command class must be named `Command` and there can only be one command per module

This is pretty much identical to who management commands are done in django.

1.5 Effects

In order to actually draw something to the screen you need to make one or multiple effects. What these effects are doing is entirely up to you. Some like to put everything into one effect and switch what they draw by flipping some internal states, but this is probably not practical for more complex things.

An effect is a class with references to resources such as shaders, geometry, fbos and textures and a method for drawing. An effect is an independent python package of specific format.

1.5.1 The Effect Package

The effect package should have the following structure (assuming our effect is named "cube").

```
cube
├── effect.py
├── shaders
│   └── cube
│       └── ...
└── textures
    └── cube
        └── ...
```

The `effect.py` module is the actual code for the effect. Directories at the same level are for local resources for the effect.

Note: Notice that the resource directories contains another sub-directory with the same name as the effect. This is because these folders are added to a **virtual directory** (for each resource type), so we should place it in a directory to reduce the chance of a name collisions.

Note: Two effects with the texture name `texture.png` in the root of their local `textures/` directory will cause a name collision were the texture from the first registered effect will be used in both effects. This can be used to override resources intentionally.

We can also decide not to have any effect-local resources and configure a project-global resource directory. More about this in *Settings*.

1.5.2 Registry

For an effect to be recognised by the system, it has to be registered in the `EFFECTS` tuple/list in your settings module. Simply add the full python path to the package. If our cube example above is located inside a `myproject` project package we need to add the string `myproject.cube`. See *Settings*.

You can always run a single effect by using the `runeffect` command.

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

If you have multiple effects, you need to crate or use an existing *Effect Managers* that will decide what effect would be active at what time or state.

1.5.3 Resources

Resource loading is baked into the effect class it self. Methods are inherited from the base `Effect` class such as `get_shader` and `get_texture`.

Remember that you can also create global resource directories for all the effects in your projects as well. This can be achieved by configuring resource finders in *Settings*.

Methods fetching resources can take additional parameters to override defaults.

Example setting texture repeat and enable anisotropic filtering:

```
self.get_texture("cube/texture.png",
                 wrap_s=GL_REPEAT, wrap_t=GL_REPEAT,
                 anisotropy=16)
```

This will also automatically generate mipmaps for the texture.

1.5.4 The Effect Module

The effect module needs to be named `effect.py` and located in the root of the effect package. It can only contain a single effect class. The name of the class doesn't matter right now, but we are considering allowing multiple effects in the future, so giving it at least a descriptive name is a good idea.

There are two important methods in an effect:

- `__init__()`
- `draw()`

The **initializer** is called before resources are loaded. This way the effects can register the resources they need. The resource managers will return an empty object that will be populated when loading starts.

The `draw` method is called by the configured *EffectManager*' (see *Effect Managers*) ever frame, or at least every frame the manager decides the effect should be active.

The standard effect example:

```
from demosys.effects import effect
from demosys.opengl import geometry
from OpenGL import GL
# from pyrr import matrix44

class DefaultEffect(effect.Effect):
    """Generated default effect"""
    def __init__(self):
        self.shader = self.get_shader("default/default.glsl")
        self.cube = geometry.cube(4.0, 4.0, 4.0)

    @effect.bind_target
    def draw(self, time, frametime, target):
        GL.glEnable(GL.GL_DEPTH_TEST)

        # Rotate and translate
        m_mv = self.create_transformation(rotation=(time * 1.2, time * 2.1, time * 0.
↪25),
                                          translation=(0.0, 0.0, -8.0))

        # Apply the rotation and translation from the system camera
```

(continues on next page)

(continued from previous page)

```
        # m_mv = matrix44.multiply(m_mv, self.sys_camera.view_matrix)

        # Create normal matrix from model-view
        m_normal = self.create_normal_matrix(m_mv)

        # Draw the cube
        with self.cube.bind(self.shader) as shader:
            shader.uniform_mat4("m_proj", self.sys_camera.projection)
            shader.uniform_mat4("m_mv", m_mv)
            shader.uniform_mat3("m_normal", m_normal)
        self.cube.draw()
```

The parameters in the draw effect is:

- `time`: The current time reported by our configured `Timer` in seconds.
- `frametime`: The time a frame is expected to take in seconds. This is useful when you cannot use time. Should be avoided.
- `target` is the target FBO of the effect

Time can potentially move at any speed or direction, so it's good practice to make sure the effect can run when time is moving in any direction.

The `bind_target` decorator is useful when you want to ensure that an FBO passed to the effect is bound on entry and released on exit. By default a fake FBO is passed in representing the window frame buffer. EffectManagers can be used to pass in your own FBOs or another effect can call `draw(..)` requesting the result to end up in the FBO it passes in and then use this FBO as a texture on a cube or do post processing.

As we can see in the example, the `Effect` base class have a couple of convenient methods for doing basic matrix math, but generally you are expected do to these calculations yourself.

1.5.5 Effect Base Class

1.5.6 Decorators

1.6 OpenGL Objects

We proved some simple and powerful wrappers over OpenGL features in the `demosys.opengl` package.

- **Texture**: Textures from images or manually constructed/generated
- **Shader**: Shader programs currently supporting vertex/fragment/geometry shaders
- **Frame Buffer Object**: Offscreen rendering targets represented as textures
- **Vertex Array Object**: Represents the geometry we are drawing using a shader

1.6.1 Texture

Textures are normally loaded by requesting the resource by path/name in the initializer of an effect using the `self.get_texture` method inherited from the `Effect` base class. We use the PIL/Pillow library to image data from file.

Textures can of course also be crated manually if needed.

1.6.2 Shader

In oder to draw something to the screen, we need a shader. There is no other way.

Shader should ideally always be loaded from `.glsl` files located in a `shaders` directory in your effect or project global resource directory. Shaders have to be written in a single file were the different shader types are separated using preprocessors.

Note: We wish to support loading shaders in other common formats such as separate files for each shader type. Feel free to make a pull request or create an issue on github.

Like textures they are loaded in the effect using the `get_shader` method in the initializer.

Once we have the reference to the shader object we will need a VAO object in order to bind it. We could of course just call `bind()`, but **the VAOs will do this for you**. More details in the VAO section below.

```
#version 410

#if defined VERTEX_SHADER
// Vertex shader here
#elif defined FRAGMENT_SHADER
// Fragment shader here
#elif defined GEOMETRY_SHADER
// Geometry shader here
#endif
```

Once the shader is bound we can set uniforms through the various `uniform_` methods.

Assuming we have a reference to a shader in `s`:

```
# Set the uniform (float) with name 'value' to 1.0
s.uniform_1f("value", 1.0)
# Set the uniform (mat4) with name `m_view' to a 4x4 matrix
s.uniform_mat4("m_view", view_matrix)
# Set the sampler2d uniform to use a Texture object we have loaded
s.sampler_2d(0, "texture0", texture)
```

The Shader class contains an internal cache of all the uniform variables the shader has, so it will generally do very efficient type checks at run time and give useful error feedback if something is wrong.

Other than setting uniforms and using the right file format for shaders, there are not much more to them.

Note: We are planning to support passing in preprocessors to shader. Please make an issue or a pull request on github.

1.6.3 Vertex Array Object

Vertex Array Objects represents the geometry we are drawing with shaders. They keep track of the buffer binding states of one or multiple Vertex Buffer Objects.

VAOs and shaders interact in a very important way. The first time the VAO and shader interacts, they will figure out if they are compatible when it comes to the attributes in the shader and the buffers in the VAO.

When we create VAOs we tell explicitly what attribute name each buffer belongs to.

Example: I have three buffers representing positions, normals and uvs.

- Map positions to `in_position` attribute with 3 components
- Map normals to `in_normal` attribute with 3 components
- Map uvs to the `in_uv` attribute with 2 components

The vertex shader will have to define the exact same attribute names:

```
in vec3 in_position;
in vec3 in_normal;
in vec2 in_uv
```

This is not entirely true. The shader will at least have to define the `in_position`. The other two attributes are optional. This is were the VAO and the Shader negotiates the attribute binding. The VAO object will on-the-fly generate a version of itself that supports the shaders attributes.

The VAO/Shader binding can also be used as a context manager as seen below, but this is optional. The context manager will return the reference to the shader so you can use a shorter name.

```
# Without context manager
vao.bind(shader)
shader.unform_1f("value", 1.0)
vao.draw()

# Bind the shader and negotiate attribute binding
with vao.bind(shader) as s:
    s.unform_1f("value", 1.0)
    # ...
# Finally draw the geometry
vao.draw()
```

When creating a VBO we need to use the *OpenGL.arrays.vbo.VBO instance* in PyOpenGL. We pass a numpy array to the constructor. It's important to use the correct `dtype` so it matches the type passed in `add_array_buffer`.

Each VBO is first added to the VAO using `add_array_buffer`. This is simply to register the buffer and tell the VAO what format it has.

The `map_buffer` part will define the actual attribute mapping. Without this the VAO is not complete.

Calling `build()` will finalize and sanity check the VAO.

The VAO initializer also takes an optional argument `mode` were you can specify what the default draw mode is. This can be overridden in `draw(mode=...)`.

The VAO will always do **very** strict error checking and give useful feedback when something is wrong. VAOs must also be assigned a name so the framework can reference it in error messages.

```
def quad_2d(width, height, xpos, ypos):
    """
    Creates a 2D quad VAO using 2 triangles.

    :param width: Width of the quad
    :param height: Height of the quad
    :param xpos: Center position x
    :param ypos: Center position y
    """
    pos = VBO(numpy.array([
        xpos - width / 2.0, ypos + height / 2.0, 0.0,
        xpos - width / 2.0, ypos - height / 2.0, 0.0,
        xpos + width / 2.0, ypos - height / 2.0, 0.0,
        xpos - width / 2.0, ypos + height / 2.0, 0.0,
```

(continues on next page)

(continued from previous page)

```
        xpos + width / 2.0, ypos - height / 2.0, 0.0,
        xpos + width / 2.0, ypos + height / 2.0, 0.0,
    ], dtype=numpy.float32))
    normals = VBO(numpy.array([
        0.0, 0.0, 1.0,
        0.0, 0.0, 1.0,
        0.0, 0.0, 1.0,
        0.0, 0.0, 1.0,
        0.0, 0.0, 1.0,
        0.0, 0.0, 1.0,
    ], dtype=numpy.float32))
    uvs = VBO(numpy.array([
        0.0, 1.0,
        0.0, 0.0,
        1.0, 0.0,
        0.0, 1.0,
        1.0, 0.0,
        1.0, 1.0,
    ], dtype=numpy.float32))
    vao = VAO("geometry:quad", mode=GL.GL_TRIANGLES)
    vao.add_array_buffer(GL.GL_FLOAT, pos)
    vao.add_array_buffer(GL.GL_FLOAT, normals)
    vao.add_array_buffer(GL.GL_FLOAT, uvs)
    vao.map_buffer(pos, "in_position", 3)
    vao.map_buffer(normals, "in_normal", 3)
    vao.map_buffer(uvs, "in_uv", 2)
    vao.build()
    return vao
```

We can also pass index/element buffers to VAOs. We can also use interleaved VBOs by passing the same VBO to `map_buffer` multiple times.

More examples can be found in the *Geometry* module.

class demosys.opengl.vao.**VAO**(*name*, *mode=4*)

Bases: object

Vertex Array Object

buffer(*buffer*, *buffer_format: str*, *attribute_names*, *per_instance=False*)

Register a buffer/vbo for the VAO. This can be called multiple times. adding multiple buffers (interleaved or not)

Parameters

- **buffer** – The buffer object. Can be ndarray or Buffer
- **buffer_format** – The format of the buffer ('f', 'u', 'i')

Returns The buffer object

draw(*shader: demosys.opengl.shader.ShaderProgram*, *mode=None*, *vertices=-1*, *first=0*, *instances=1*)

Draw the VAO. Will use `glDrawElements` if an element buffer is present and `glDrawArrays` if no element array is present.

Parameters

- **shader** – The shader to draw with
- **mode** – Override the draw mode (GL_TRIANGLES etc)

- **vertices** – The number of vertices to transform
- **first** – The index of the first vertex to start with
- **instances** – The number of instances

index_buffer (*buffer*, *index_element_size=4*)
Set the index buffer for this VAO

Parameters

- **buffer** – Buffer object or ndarray
- **index_element_size** – Byte size of each element. 1, 2 or 4

transform (*shader*, *buffer: moderngl.Buffer*, *mode=None*, *vertices=-1*, *first=0*, *instances=1*)
Transform vertices. Stores the output in a single buffer.

Parameters

- **buffer** – The buffer to store the output
- **mode** – Draw mode (for example *POINTS*
- **vertices** – The number of vertices to transform
- **first** – The index of the first vertex to start with
- **instances** – The number of instances

Returns

1.6.4 Frame Buffer Object

Frame Buffer Objects are offscreen render targets. Internally they are simply textures that can be used further in rendering. FBOs can even have multiple layers so a shader can write to multiple buffers at once. They can also have depth/stencil buffers. Currently we use use a depth 24 / stencil 8 buffer by default as the depth format.

Creating an FBO:

```
# Shorcut for creating a single layer FBO with depth buffer
fbo = FBO.create(1024, 1024, depth=True)

# Multilayer FBO (We really need to make a shortcut for this!)
fbo = FBO()
fbo.add_color_attachment(texture1)
fbo.add_color_attachment(texture2)
fbo.add_color_attachment(texture3)
fbo.set_depth_attachment(depth_texture)

# Binding and releasing FBOs
fbo.bind()
fbo.release()

# Using a context manager
with fbo:
    # Draw stuff in the FBO
```

When binding the FBOs with multiple color attachments it will automatically call `glDrawBuffers` enabling multiple outputs in the fragment shader.

Shader example with multiple layers:

```
#version 410

layout(location = 0) out vec4 outColor0;
layout(location = 1) out vec4 outColor1;
layout(location = 2) out vec4 outColor2;

void main( void ) {
    outColor0 = vec4(1.0, 0.0, 0.0, 1.0)
    outColor1 = vec4(0.0, 1.0, 0.0, 1.0)
    outColor1 = vec4(0.0, 0.0, 1.0, 1.0)
}
```

Will draw red, green and blue in the separate layers in the FBO/textures.

Warning: It's important to use explicit attribute locations as not all drivers will guarantee preservation of the order and things end up in the wrong buffers!

Another **very important** feature of the FBO implementation is the concept of FBO stacks.

- The default render target is the window frame buffer.
- When the stack is empty we are rendering to the screen.
- When binding an FBO it will be pushed to the stack and the correct viewport for the FBO will be set
- When releasing the FBO it will be popped from the stack and the viewport for the default render target will be applied
- This also means we can build deeper stacks with the same behavior
- The maximum stack depth is currently 8 and the framework will aggressively react when FBOs are popped and pushed in the wrong order

A more complex example:

```
# Push fbo1 to stack, bind and set viewport
fbo1.bind()
# Push fbo2 to stack, bind and set viewport
fbo2.bind()
# Push fbo3 to stack, bind and set viewport
fbo3.bind()
# Pop fbo3 from stack, bind fbo2 and set the viewport
fbo3.release()
# Pop fbo2 from stack, bind fbo1 and set the viewport
fbo2.release()
# Pop fbo1 from stack, unbind the fbo and set the screen viewport
fbo1.release()
```

Using context managers:

```
with fbo1:
    with fbo2:
        with fbo2:
            pass
```

This is especially useful in realation to the `draw` method in effects. The last parameter is the target FBO. The effect will never know if the FBO passed in is the fake window FBO or an actual FBO. It might also do offscreen rendering to its own fbos and things start get get really ugly.

The FBO stack makes this fairly painless.

By using the `bind_target` decorator on the `draw` method of your effect you will never need to think about this issue. Not having to worry about resporting the viewport size is also a huge burden off our shoulders.

```
@effect.bind_target
def draw(self, time, frametime, target):
    # ...
```

There are of course ways to bypass the stack, but should be done with extreme caution.

Note: We are also aiming to support layered rendering using the geometry shader. Please make an issue or pull request on github.

---

class demosys.opengl.fbo.**FBO**

Bases: object

Frame buffer object

clear(*red=0.0*, *green=0.0*, *blue=0.0*, *alpha=0.0*, *depth=1.0*)

Clears the FBO using `glClear`.

static create(*size*, *components=4*, *depth=False*, *dtype='f1'*, *layers=1*)

Create a single or multi layer FBO

Parameters

- **size** – (tuple) with and height
- **components** – (tuple) number of components. 1, 2, 3, 4
- **depth** – (bool) Create a depth attachment
- **dtype** – (string) data type per r, g, b, a ...
- **layers** – (int) number of color attachments

Returns A new FBO

static create_from_textures(*color_buffers: List[demosys.opengl.texture.Texture2D]*, *depth_buffer: demosys.opengl.texture.DepthTexture = None*)

Create FBO from existing textures

Parameters

- **color_buffers** – List of textures
- **depth_buffer** – Depth texture

Returns FBO instance

draw_color_layer(*layer=0*, *pos=(0.0, 0.0)*, *scale=(1.0, 1.0)*)

Draw a color layer in the FBO. :param layer: Layer ID :param pos: (tuple) offset x, y :param scale: (tuple) scale x, y

draw_depth(*near*, *far*, *pos=(0.0, 0.0)*, *scale=(1.0, 1.0)*)

Draw a depth buffer in the FBO.

Parameters

- **near** – projection near.
- **far** – projection far.
- **pos** – (tuple) offset x, y

- **scale** – (tuple) scale x, y

read(*viewport=None*, *components=3*, *attachment=0*, *alignment=1*, *dtype='f1'*) → bytes
Read the content of the framebuffer.

Parameters

- **viewport** – (tuple) The viewport
- **components** – The number of components to read.
- **attachment** – The color attachment
- **alignment** – The byte alignment of the pixels
- **dtype** – (str) dtype

release(*stack=True*)
Bind FBO popping it from the stack

Parameters **stack** – (bool) If the bind should be popped form the FBO stack.

size
Attempts to determine the pixel size of the FBO. Currently returns the size of the first color attachment. If the FBO has no color attachments, the depth attachment will be used. Raises `FBOError if the size cannot be determined.

Returns (w, h) tuple representing the size in pixels

stack = []

use(*stack=True*)
Bind FBO adding it to the stack.

Parameters **stack** – (bool) If the bind should push the current FBO on the stack.

1.7 Timers

Timers are classes keeping track of time passing the value to the effect's `draw` methods. We should assume that time can move in any direction at any speed. Time is always reported as a float in seconds.

The default timer if not specified in settings:

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

This is a simple timer starting at 0 when effects start drawing. All timers should respond correctly to pause `SPACE`.

1.7.1 Standard Timers

- `demosys.timers.Timer`: Default timer just tracking time in seconds
- `demosys.timers.Music`: Timer playing music reporting duration in the song
- `demosys.timers.RocketTimer`: Timer using the rocket sync system
- `demosys.timers.RocketMusicTimer`: Timer using the rocket sync system with music playback

1.7.2 Custom Timer

You create a custom timer by extending `demosys.timers.base.BaseTimer`.

1.8 Effect Managers

An effect manager is responsible of:

- Instantiating effects
- Knowing what effect should be drawn based in some internal state
- Reading keyboard events if this is needed (optional)

You are fairly free to do what you want. Having control over effect instantiation also means you can make multiple instances of the same effect and assign different resources to them.

The most important part in the end is how you implement `draw()`.

Some sane or insane examples to get started:

- Simply hard code what should run at what time or state
- A manger that cycles what effect is active based on a next/previous key
- Cycle effects based on a duration property you assign to them
- Load some external timer data describing what effect should run at what time. This can easily be done with rocket (we are planning to make a manager for this)
- You could just put all your draw code in the manager and not use effects
- Treat the manager as the main loop of a simple game

This is an example of the default `SingleEffectManager`.

```
class SingleEffectManager(BaseEffectManger):
    """Run a single effect"""
    def __init__(self, effect_module=None):
        """
        Initalize the manager telling it what effect should run.

        :param effect_module: The effect module to run
        """
        self.active_effect = None
        self.effect_module = effect_module

    def pre_load(self):
        """
        Initialize the effect that should run.
        """
        # Instantiate all registered effects
        effect_list = [cfg.cls() for name, cfg in effects.effects.items()]
        # Find the single effect we are supposed to draw
        for effect in effect_list:
            if effect.name == self.effect_module:
                self.active_effect = effect

        # Show some modest anger when we have been lied to
        if not self.active_effect:
            print("Cannot find effect '{}'".format(self.active_effect))
            print("Available effects:")
            print("\n".join(e.name for e in effect_list))
            return False
        return True
```

(continues on next page)

(continued from previous page)

```
    def post_load(self):
        return True

    def draw(self, time, frametime, target):
        """This is called every frame by the framework"""
        self.active_effect.draw(time, frametime, target)

    def key_event(self, key, scancode, action, mods):
        """Called on most key presses"""
        print("SingleEffectManager:key_event", key, scancode, action, mods)
```

It's important to understand that `pre_load` is called before resources are loaded and this is the correct place to instantiate effects. `post_load` is called right after loading is done.

The `draw` method is called every frame and you will have to send this to the effect you want to draw.

The `key_events` method will trigger on key presses.

1.8.1 BaseEffectManger

1.9 Geometry

The `demosys.opengl.geometry` module currently provides some simple functions to generate VAOs.

- Quad: Full screen quads for drawing offscreen buffers
- Cube: Cube with normals, uvs and texture coordinates
- Plane: A plane with a dimension and resolution
- Points: Random points in 3D

Note: We definitely need more here. Please make pull requests or make an issue on github.

1.9.1 Scene/Mesh File Formats

The `demosys.scene.loaders` are meant for this.

Note: We currently do not support loading any formats. If you have any suggestions in this area, please make an issue on github.

1.9.2 Generating Custom Geometry

To efficiently generate geometry in Python we must avoid as much memory allocation as possible. As mentioned in other sections we use PyOpenGL's VBO class that takes numpy arrays. We also use pyrr for vector/matrix math/representation.

Note: This is a "best practices" guide to efficiently generate geometry with python code that will scale well even for large amounts of data. This was benchmarked generating various vertex formats with 1M vertices. For fairly small data sizes doesn't matter that much.

The naive way of generating geometry would probably look something like this:

```
from OpenGL import GL
from OpenGL.arrays.vbo import VBO
import numpy
from pyrr import Vector3

def random_points(count):
    points = []
    for p in range(count):
        # Let's pretend we calculated random values for x, y, z
        points.append(Vector3([x, y, x]))

    # Create VBO enforcing float32 values with numpy
    points_vbo = VBO(numpy.array(points, dtype=numpy.float32))

    vao = VAO("random_points", mode=GL.GL_POINTS)
    vao.map_array_buffer(GL.GL_FLOAT, points_vbo)
    vao.map_buffer(points_vbo, "in_position", 3))
    vao.build()
    return vao
```

This works perfectly fine, but we allocate a new list for every iteration and pyrr internally creates a numpy array. The `points` list will also have to dynamically expand. This gets exponentially more ugly as the `count` value increases.

We move on to version 2:

```
def random_points(count):
    # Pre-allocate a list containing zeros of length count * 3
    points = [0] * count * 3
    # Loop count times incrementing by 3 every frame
    for p in range(0, count * 3, 3):
        # Let's pretend we calculated random values for x, y, z
        points[p] = x
        points[p + 1] = y
        points[p + 2] = z

  points_vbo = VBO(numpy.array(points, dtype=numpy.float32))
```

This version is orders of magnitude faster because we don't allocate memory in the loop. It has one glaring flaw though. It's **not a very pleasant read** even for such simple task, and it will not get any better if we add more complexity.

Let's move on to version 3:

```
def random_points(count):
    def generate():
        for p in range(count):
            # Let's pretend we calculated random values for x, y, z
            yield x
            yield y
            yield z

    points_vbo = VBO(numpy.fromiter(generate(), count=count * 3, dtype=numpy.float32)
```

Using generators in Python like this is much a cleaner way. We also take advantage of numpy's `fromiter()` that basically slurps up all the numbers we emit with yield into its internal buffers. By also telling numpy what the final size of the buffer will be using the `count` parameter, it will pre-allocate this not having to dynamically increase it's internal buffer.

Generators are extremely simple and powerful. If things get complex we can easily split things up in several functions because Python's `yield from` can forward generators.

Imagine generating a single VBO with interleaved position, normal and uv data:

```
def generate_stuff(count):
    # Returns a distorted position of x, y, z
    def pos(x, y, z):
        # Calculate..
        yield x
        yield y
        yield x

    def normal(x, y, z):
        # Calculate
        yield x
        yield y
        yield z

    def uv(x, y, x):
        # Calculate
        yield u
        yield v

    def generate(count):
        for i in range(count):
            # resolve current x, y, z pos
            yield from pos(x, y, z)
            yield from normal(x, y, z)
            yield from uv(x, y, z)

    interleaved_vbo = VBO(numpy.fromiter(generate(), count=count * 8, dtype=numpy.
↪float32)
```

1.9.3 The geometry Module

1.10 Audio

We currently use pygame's mixer module for music playback. More work needs to be done to find a better alternative as depending on such a huge package should not be needed.

You will have to manually add pygame to your requirements and pip install the package.

In oder to get pygame to work you probably need sdl, sdl_mixer and libvorbis. These are binary dependencies and not python packages.

We need to figure out what requiremnets are actually needed.

As mentioned in readme, the state of audio is not in good shape.

The sound player an be a bit wonky at times on startup refusing to play on some platforms. We have tried a few libraries and ended up using pygame's mixer module. (Optional setup for this)

Audio Requirements:

- As the current position in the music is what all draw timers are based on, we need a library that can deliver very accurate value for this.
- Efficient and accurate seeking + pause support
- Some way to extract simple data from the music for visualisation

1.11 Matrix and Vector math with pyrr

Pyrr has both a procedural and object oriented api.

See pyrr for official docs.

Note: We should probably add some more examples here. Feel free to make an issue or pull request on github.

1.11.1 Examples

Identity

```
# procedural
>> m = matrix44.create_identity()
>> print(m)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])

# object
>> m = Matrix44.identity()
>> print(m)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

Matrices produced by `Matrix44` are also just numpy arrays as the class extends `numpy.ndarray`. We can pretty much use the APIs interchangeably unless we rely on a method in the class. They can both be passed right into shaders as matrix uniforms.

Rotation

```
# Short version
mat = Matrix44.from_eulers(Vector3(rotation))

# Long version
rot_x = matrix44.create_from_x_rotation(rotation[0])
rot_y = matrix44.create_from_y_rotation(rotation[1])
rot_z = matrix44.create_from_z_rotation(rotation[2])
mat = matrix44.multiply(x, y)
mat = matrix44.multiply(mat, z)
```

1.11.2 Covert

```
# mat4 to mat3
mat3 = Matrix33.from_matrix44(mat)
# mat3 to mat4
mat4 = Matrix44.from_matrix33(mat)
```

1.11.3 Common Mistakes

Matrices and vectors are just numpy arrays. When multiplying matrices, use the `mult` method/function.

```
mat = matrix44.mult(mat1, mat2)
```

Using the $\star$ operator would just make a product of the two arrays.

1.12 Performance

When using a high level language such as Python for real time rendering we must be extra careful with the total time we spend in Python code every frame. At 60 frames per second we only have 16 milliseconds to get the job done. This is ignoring delays or blocks caused by OpenGL calls.

Note: How important performance is will of course depend on the project. Visualization for a scientific application doing some heavy calculations would probably not need to run at 60 fps. It's also not illegal to not care about performance.

Probably the biggest enemy to performance in python is **memory allocation**.

Try to avoid creating new objects every frame if possible. This includes all mutable data types such as lists, sets, dicts.

Another area is updating buffer object data such as VBOs and Textures. If these are of a fairly small size it might not be a problem, but do not expect pure Python code to be able to efficiently feed CPU-generated data to OpenGL. If this data comes from a library though ctypes and we can avoid re-allocating memory for each frame we might be good, but this is not always easy to determine and will needs testing.

Try to do as much as possible on the GPU. Use features like transform feedback to alter buffer data and use your creativity to find efficient solutions.

Performance in rendering is not straight forward to measure in any language. Simply adding timers in the code will not really tell us much unless we also query OpenGL about the performance.

We could also try to compile your project with pypy, but we have not tested this (yet).

We can also strive to do more with less. Rendering, in the end, is really just about creating illusions.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index