demosys-py Documentation

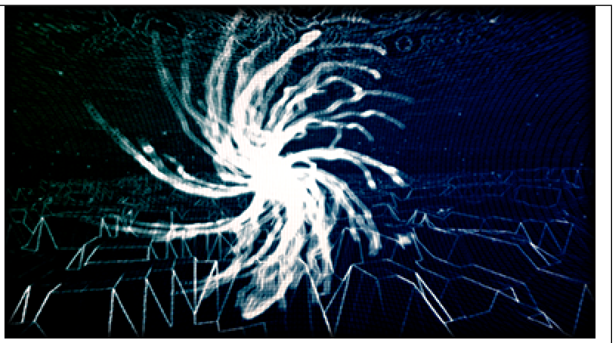
Release 1.0.7

Einar Forselv

Aug 04, 2018

Contents:

A Python 3 cross platform OpenGL 3.3+ core framework based on ModernGL



CHAPTER 1

Getting Started

1.1 Python 3

Make sure you have Python 3.5 or later installed. On Windows and OS X you can simply install the latest Python 3 by downloading an installer from the official Python site.

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.

1.3 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.

1.4 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.

1.5 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
```

CHAPTER 2

Reference

2.1 Effect

The base Effect class extended in effect modules.

2.1.1 Draw Methods

2.1.2 Resource Methods

2.1.3 Utility Methods

2.1.4 Attributes

2.1.5 Decorators

2.2 Texture2D

2.2.1 Create

2.2.2 Methods

2.2.3 Attributes

2.3 TextureArray

2.3.1 Create

2.3.2 Methods

2.3.3 Attributes

2.4 DepthTexture

2.4.1 Create

2.4.2 Methods

2.4.3 Attributes

2.5 FBO

2.5.1 Create

2.5.2 Methods

2.5.3 Attributes

2.6 VAO

2.6.1 Methods

2.6.2 Draw Methods

2.7 ShaderProgram

2.7.1 Method

2.8.1 Functions

CHAPTER 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.

3.1 OPENGL

Sets the minimum required OpenGL version to run your project. A forward compatible core context will be always be created. This means the system will pick the highest available OpenGL version available.

The default and lowest OpenGL version is 3.3 to support a wider range of hardware. To make your project work on OS X you cannot move past version 4.1 (sadly).

```
OPENGL = {
    "version": (3, 3),
}
```

Only increase the OpenGL version if you use features above 3.3.

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 (buffer size will nomally be 2 x the window size)

```
WINDOW = {
    "class": "demosys.context.glfw.GLFW_Window",
    "size": (1280, 768),
    "aspect_ratio": 16 / 9,
    "fullscreen": False,
```

(continues on next page)

(continued from previous page)

```
    "resizable": False,
    "vsync": True,
    "title": "demosys-py",
    "cursor": False,
}
```

- `class`: The class for our window (This can be customized)
- `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)
- 24 bit depth buffer
- Double buffering
- color and depth buffer is cleared for every frame

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')
```

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 audio file.
- `demosys.timers.RocketTimer` is the same as the default timer, but uses the pyrocket library with options to connect to an external sync tracker.
- `demosys.timers.RocketMusicTimer` requires `MUSIC` and `ROCKET` to be configured.

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

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 (xml file)
- `files`: The absolute path to the directory containing binary track data

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

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',
)
```

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.

3.8 SHADER_STRICT_VALIDATION

Boolean value. If `True` shaders will raise `ShaderError` when setting uniforms variables that don't exist.

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.

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',
)
```

3.10 TEXTURE_DIRS/FINDERS

Same principle as `SHADER_DIRS` and `SHADER_FINDERS`. The `EffectDirectoriesFinder` will look for a `textures` directory in effects.

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

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

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. The `EffectDirectoriesFinder` will look for a `scenes` directory

```
# Absolute path to a project-global scene directory
SCENE_DIRS = (
```

(continues on next page)

(continued from previous page)

```
    os.path.join(PROJECT_DIR, 'resources/scenes'),
)

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

3.12 SCREENSHOT_PATH

Absolute path to the directory screenshots will be saved. If not defined or the directory don't exist it will be created.

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

CHAPTER 4

Controls

4.1 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)

4.2 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

CHAPTER 5

Guides

5.1 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 and a `manage.py` 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 the entrypoint for running your project

5.1.1 Effects

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`.

5.1.2 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.

Examples of `manage.py` usage:

```
# Create effect inside a project
python manage.py createeffect myproject/myeffect
# Run a specific effect
python manage.py runeffect myproject.myeffect
# Run using the configured effect manager
python manage.py run
# Run a cusom command
python manage.py <cusom command>
```

5.1.3 Effect Templates

A collection of effect templates reside in `effect_templates` directory. To list the available templates:

```
$ ./manage.py createeffect --template list
Available templates: cube_simple, sphere_textured, raymarching_simple
```

To create a new effect with a specific template

```
$ ./manage.py createeffect myproject/myeffect --template raymarching_simple
```

Note: If you find the current effect templates insufficent please make a pull request or report the issue on github.

5.1.4 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")
```

(continues on next page)

(continued from previous page)

```
    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.

5.2 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.

5.2.1 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 directory/package. This is because these folders are by default added to a project wide search path (for each resource type), so we should place it in a directory to reduce the chance of a name collisions.

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

5.2.2 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.

5.2.3 Resources

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

Methods fetching resources can take additional parameters to override defaults.

```
# Generate mipmaps for the texture
self.get_texture("cube/texture.png", mipmap=True)
```

5.2.4 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. An opengl context should exist.

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:

```
import moderngl as mgl
from demosys.effects import effect
from demosys import geometry
# from pyrr import matrix44


class SimpleCubeEffect(effect.Effect):
    """Generated default effect"""
    def __init__(self):
        self.shader = self.get_shader("cube_plain.glsl", local=True)
        self.cube = geometry.cube(4.0, 4.0, 4.0)

    @effect.bind_target
    def draw(self, time, frametime, target):
        self.ctx.enable(mgl.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))
```

(continues on next page)

(continued from previous page)

```
        # Apply the rotation and translation from the system camera
        # 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
        self.shader.uniform("m_proj", self.sys_camera.projection.tobytes())
        self.shader.uniform("m_mv", m_mv.astype('f4').tobytes())
        self.shader.uniform("m_normal", m_normal.astype('f4').tobytes())
        self.shader.uniform("time", time)
        self.cube.draw(self.shader)
```

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.
- `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.

5.3 Effect Managers

An effect manager is responsible for:

- Instantiating effects
- Knowing what effect should be drawn based in some internal state
- Reading input 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`. Like the name suggests, it runs a single effect only.

```
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

    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.

5.3.1 BaseEffectManger

5.4 The geometry module

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

Examples:

```
from demosys import geometry
# Create a fullscreen quad for overing the entire screen
vao = geometry.quad_fs()
# Create a 1 x 1 quad on the xy plane
vao = geometry.quad_2f(with=1.0, height=1.0)
# Create a unit cube
vao = geometry.cube(1.0, 1.0, 1.0)
# Create a bounding box
vao = geometry.bbox()
# Create a sphere
vao = geometry.sphere(radius=0.5, sectors=32, rings=16)
# Random 10.000 random points in 3d
vao = geometry.points_random_3d(10_000)
```

Note: Improvements or suggestions can be made by through pull requests or issues on github.

See the `geometry` reference for more info.

5.4.1 Scene/Mesh File Formats

The `demosys.scene.loaders` currently support loading wavefront obj files and gltf.

You can create your own scene loader by adding the loader class to `SCENE_LOADERS`.

```
SCENE_LOADERS = (
    'demosys.scene.loaders.gltf.GLTF2',
    'demosys.scene.loaders.wavefront.ObjLoader',
)
```

5.4.2 Generating Custom Geometry

To efficiently generate geometry in Python we must avoid as much memory allocation as possible. If performance doesn't matter, then take this section lightly. Lbraries like `numpy` can also be used to generate geometry.

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

```
import numpy
import moderngl
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_data = numpy.array(points, dtype=numpy.float32)

    vao = VAO("random_points", mode=moderngl.POINTS)
    vao.buffer(points_data, 'f4', "in_position")
    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_data = 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. 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_data = 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 its 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
```

(continues on next page)

(continued from previous page)

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

5.5 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`.

5.5.1 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

5.5.2 Custom Timer

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

5.6 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.

5.6.1 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)
```

5.6.2 Covert

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

5.6.3 Common Mistakes

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

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

Using the `*` operator would just make a product of the two arrays.

5.7 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.

5.8 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

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index