demosys-py Documentation

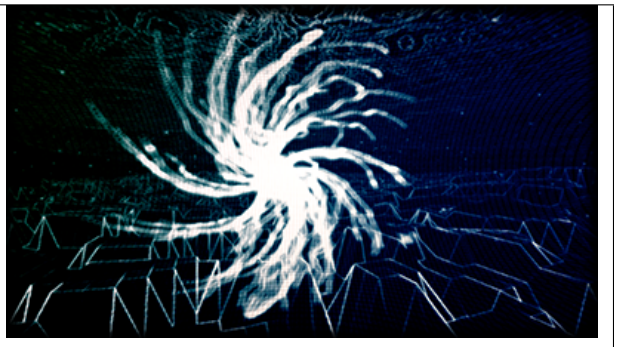
Release 2.0.3

Einar Forselv

Dec 29, 2018

Getting Started

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



CHAPTER 1

Getting Started

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.

1.1 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 and activated an isolated Python environment and are ready to install packages without affecting the Python versions in our operating system.

1.2 Setting up a Project

Install the demosys-py

```
pip install demosys-py
```

This 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
└── project.py
manage.py
```

- `settings.py`: the settings for your project
- `project.py`: your project config
- `manage.py`: entrypoint script for running your project

These files can be left unchanged for now. We mainly need `manage.py` as an entrypoint to the framework and the default `settings` should be enough.

- An overview of the settings can be found in the /reference/settings section.
- More information about projects can be found in the *Project* section.

1.3 Creating an Effect Package

In order to draw something to the screen we have to make an effect package with at least one effect. We can create this effect package in the root or inside `myproject`. Since we don't care about project (yet), we create it in the root.

```
demosys-admin createeffect cube
```

We should now have the following structure:

```
cube
├── effects.py
├── dependencies.py
└── resources
    └── programs
        └── cube
            └── default.glsl
```

The `cube` directory is a copy of the deault effect pacakge template:

- The `effects.py` module containing one or multiple *`demosys.effects.Effect`* implementation
- A `dependencies.py` module describing effect package dependencies and resources for this package
- A local `resources/programs` directory for glsl shader programs specific to the effect

dependencies.py:

```
from demosys.resources.meta import ProgramDescription

# We don't depend on any other effect packages at the moment
effect_packages = []

# We tell the system to load our shader program storing it with label "cube_plain".
# The shader program can then be obtained in the effect instance using this label.
resources = [
    ProgramDescription(label='cube_plain', path='cube_plain.glsl'),
]
```

Other resource types are also supported such as textures, programs, scenes/meshes and miscellaneous data types. More on this in the /user_guide/resources section.

Also take a minute to look through the `effects.py` module. It contains a fair amount of comments what will explain things. This should be very recognizalbe if you have worked with OpenGL.

Note: Notice the `programs` directory also has a sub-folder with the same name as the effect package. This is because these directories are added to a search path for all programs and the only way to make these resources unique is to put them in a directory.

We can now run the effect that shows a spinning cube

```
python manage.py runeffect cube
```

Effect packages can be reusable between projects and can also potentially be shared with others as python packages in private repos or on Python Package Index.

CHAPTER 2

User Guide

2.1 Effects

In order to actually render something to the screen you need to make one or multiple effects. What these effects are doing is entirely up to you. Effects have methods for fetching loaded resources and existing effect instances. Effects can also create new instances of effects if needed. This would happend during initialization.

Effect examples can be found in the examples directory in the root of the repository.

A bascic effect would have the following structure:

```
from demosys.effects import Effect

class MyEffect(Effect):

    def __init__(self):
        # Do initialization here

    def draw(self, time, frametime, target):
        # Called every frame the effect is active
```

2.1.1 The Effect Package

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

```
cube
├── effects.py
├── dependencies.py
└── resources
    └── programs
        └── cube
            └── cube.glsl
    └── textures
```

(continues on next page)

(continued from previous page)

```
    └── scenes
    └── data
```

The `effects.py` module can contain one or multiple effects. The effect package can also have no effects and all and only provide resources for other effects to use. The `effects.py` module is still required to be present.

2.1.2 Dependencies

The `dependencies.py` module is required to be present. It describes its own resources and what effect packages it may depend on.

Example:

```
from demosys.resources.meta import ProgramDescription

effect_packages =
    'full.python.path.to.another.package',
]

resources = [
    ProgramDescription(label='cube_plain', path='cube_plain.glsl'),
]
```

Resources are given labels and effects can fetch them by this label. When adding effect package dependencies we make the system aware of this package so their resources are also loaded. The effects in the depending package will also be registered in the system and can be instantiated.

2.1.3 Resources

The `resources` directory contains fixed directory names where resources of specific types are supposed to be located. When an effect package is loaded, paths to these directories are added so the system can find them.

Note: Notice that the resource directories contains another sub-directory with the same name as the effect 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.

Having resources in the effect package itself is entirely optional. Resources can be located anywhere you want as long as you tell the system where they can be found. This is covered in *Settings*.

Reasons to have resources in effect packages is to create an independent resuable effect package you could even distribute. Also when a project grows with lots of effect packages it can be nice to keep effect specific resources separate.

We currently support the following resource types:

- Shader programs
- Scene/mesh data (glfw 2.0 or wavefront obj)
- Textures (loaded with Pillow)
- Data (generic data loader supporting binary, text and json)

We load these resources by creating resource description instances:

```
from demosys.resources.meta import (TextureDescription,
                                    ProgramDescription,
                                    SceneDescription,
                                    DataDescription)

# Resource list in effect package or project
resources = [
    # Textures
    TextureDescription(label="bricks", path="bricks.png"),
    TextureDescription(label="wood", path="bricks.png", mipmap=True),

    # Shader programs
    ProgramDescription(label="cube_plain", path="cube_plain.glsl"),
    ProgramDescription(
        label="cube_textured",
        vertex_shader="cube_textured.vs",
        fragment_shader="cube_textured.fs"
    ),

    # Scenes / Meshes
    SceneDescription(label="cube", path="cube.obj"),
    SceneDescription(label="sponza", path="sponza.gltf"),
    SceneDescription(label="test", path="test.glb"),

    # Generic data
    DataDescription(label="config", path="config.json", loader="json"),
    DataDescription(label="rawdata", path="data.dat", loader="binary"),
    DataDescription(label="random_text", path="info.txt", loader="text"),
]
```

The Effect base class have methods avaiable for fetching loaded resources by their label. See the *demosys.effects.Effect*.

There are no requirements to use the resource system, but it provides a convenient way to ensure resources are only loaded once and are loaded and ready before effects starts running. If you prefer to open files manually in an effect initializer with `open` you are free to do that.

You can also load resources directly at an point in time by using the `resources` package:

```
from demosys.resources import programs, textures, scenes, data
from demosys.resources.meta import (TextureDescription,
                                    ProgramDescription,
                                    SceneDescription,
                                    DataDescription)

program = programs.load(ProgramDescription(label="cube_plain", path="cube_plain.glsl
↪"))
texture = textures.load(TextureDescription(label="bricks", path="bricks.png"))
scene = scenes.load(SceneDescription(label="cube", path="cube.obj"))
config = data.load(DataDescription(label="config", path="config.json", loader="json"))
```

This is not recommended, but in certain instances it can be unavoidable. An example could be loading a piece of data that references other resources. These are common to use in resource loader classes. Also, if you for some reason need to load something while effects are already, this would be the solution.

2.1.4 Running an Effect Package

Effect packages can be run by using the `runeffect` command:

```
python manage.py runeffect <python.path.to.package>

# Example
python manage.py runeffect examples.cubes
```

This will currently look for the first effect class with the `runnable` attribute set to `True`, make an instance of that effect and call its `draw` method every frame. The effect package dependencies are also handled. (All handled by `DefaultProject` class)

The runnable effect is resposible for instantiating other effects it depends on and call them directly.

Optionally you can also specify the exact effect to run in the effect package by adding the class name to the path:

```
python manage.py runeffect <python.path.to.package>.<effect class name>

# Example
python manage.py runeffect examples.cubes.Cubes
```

If you need a more complex setup where multiple runnable effects are involved, you need to create a proper project config using `project.py` and instead use the `run` command.

2.2 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 is also required to run effect packages using `runeffect`.

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
project.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
- `project.py` is used to initialize more complex project.

2.2.1 The project.py Module

The `project.py` module is the standard location to configure more complex projects. We achieve this by creating a class implementing `BaseProject`. This class contains references to all resources, effect packages, effect instances and whatnot so we can freely configure our project:

```
from demosys.project.base import BaseProject

class Project(BaseProject):
    effect_packages = [
        'myproject.cube',
    ]
    resources = []

    def create_effect_instances(self):
        # Create three instances of a cube effect that takes a color keyword argument
        # adding them to the internal effect instance dictionary using label as the␣
↪key
        # Args: label, class name, arguments to effect initializer
        self.create_effect('cube_red', 'CubeEffect', color=(1.0, 0.0, 0.0))
        self.create_effect('cube_green', 'CubeEffect', color=(0.0, 1.0, 0.0))
        # Use full path to class
        self.create_effect('cube_blue', 'myproject.cube.CubeEffect', color=(0.0, 0.0,␣
↪1.0))
```

This project configuration is used when the `run` command is issued. For the project class to be recognized we need to update the `settings.PROJECT` attribute with the python path:

```
PROJECT = 'myproject.project.Project'
```

`manage.py run` will now run the project using this project configuration.

How you organize your resources and effects are entirely up to you. You can load all resources in the Project class and/or have effect packages loading their own resources. Resources dependencies for effect packages are always loaded automatically when adding the package to `effect_packages` (can be overriden by implementing the `create_external_resources` method.

The Project class also have direct access to the moderngl context through `self.ctx`, so you are free to manually create any global resource (like framebuffers) and assign them to effects.

The created effect instances can then be used by a timeline class deciding what effects should be rendered at any given point in time. The default timeline configured just grabs the first runnable effect it finds and render only that one.

2.2.2 Timers

Timers are classes responsible for controlling the current time. It simply reports the number of seconds as a float since effect rendering started. Timers also need to support pausing and time seeking so we can freely move around in the timeline.

This time value is passed through the configured timeline class and forwarded to each active effect through their `draw()` method. We should assume time can move in any direction at any speed and suddenly jump forward and backwards in time.

The default timer if not specified in settings:

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

Standard Timers

- *demosys.timers.clock.Timer*: Default timer just tracking time in seconds using pythons `time` module.

- *demosys.timers.music.Timer*: Timer playing music reporting duration in the song
- *demosys.timers.rocket.Timer*: Timer using the rocket sync system
- *demosys.timers.rocketmusic.Timer*: Timer using the rocket sync system with music playback

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

2.2.3 Timelines

A timeline is a project responsible for knowing exactly when an effect instance is active based on the reported time from a timer.

The current standard timelines are:

- `demosys.timeline.single.Timeline`: Grabs a the single effect instance from your project rendering it
- `demosys.timeline.rocket.Timeline`: The active status of each effect is decided by rocket

New timeline classes can be created by extending `demosys.timeline.base.BaseTimeline`.

2.2.4 Effects Package Organization

By default it's a good idea to put effect packages inside the project package as this protects you from package name collisions and makes distribution of the project through Python Package Index simpler.

An exeption is when creating a reusable effect package in a separate repository. The effect package could for example be named `demosys_postprocessing` containing configurable effects doing various postprocessing techniques.

Note: We encourage you to share reusable effect packages on pypi and are planning add these links these in the project description on github. (Make an issue or PR)

2.2.5 manage.py

The `manage.py` script is an alternative entry point to `demosys-admin` that properly setting the `DEMOSYS_SETTINGS_MODULE` enviroment variable for your project. The main purpose of `demosys-admin` is to initially have an entry point to the commands creating a projects when we don't have a `manage.py` yet.

By default `manage.py` sets your settings module to `<project_name>.settings` matching the default auto generated settings module. You can override this by setting the `DEMOSYS_SETTINGS_MODULE` enviroment variable before running `manage.py`.

Examples of `manage.py` usage:

```
# Create a new project
python manage.py createproject myproject

# Create effect inside a project
python manage.py createeffect myproject/myeffect

# Run a specific effect package
python manage.py runeffect myproject.myeffectpackage

# Run using the ``project.py`` configuration.
```

(continues on next page)

(continued from previous page)

```
python manage.py run

# Run a cusom command
python manage.py <custom command> <custom arguments>
```

The `manage.py` script is executable by default and can be executed directly `./manage.py <arguments>` on linux and OS X.

2.2.6 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, empty, 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.

2.2.7 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 `convert_meshes`.

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

```
myproject
└── management
    └── commands
        └── convert_meshes.py
```

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

```
./manage.py convert_meshes
```

Our test command would look like this:

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

class Command(BaseCommand):
    help = "Converts meshes to a more desired format"

    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

The idea is to create modules doing the actual command work in the `management` package while the command modules deal with the basic input/output.

2.3 Creating Geometry

In order to render something to the screen we need geometry as vertex arrays.

We have the following options:

- Using the *demosys.geometry* module (or extend the geometry module)
- Loading scenes/meshes from file using the supported file formats (or create new loaders of other formats)
- Generating your own geometry programmatically

2.3.1 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 *demosys.geometry* reference for more info.

2.3.2 Scene/Mesh File Formats

The `demosys.scene.loaders` currently support loading wavefront obj files and gltf 2.0 files.

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',
)
```

2.3.3 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.

There are many lbraries out there such as `numpy` capable of generating geometry fairly efficiently. Here we mainly focus on creating it ourselves using pure python code. We're also using the *demosys.opengl.vao.VAO* for vertex buffer construction. This can easily be translated into using `moderngl.VertexArray` directly if needed.

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 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 at least and order 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 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
        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)
```

2.4 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.

2.4.1 Your Worst Enemy: Memory Allocation

Probably the biggest enemy to performance in python is **memory allocation**. Try to avoid creating new objects when possible.

Try to do as much as possible on the GPU. Tru to use features like transform feedbacks, instancing and indirect rendering. Use your creativity to find efficient solutions.

When doing many draw calls, do as little as possible between those draw calls. Doing matrix math in python with numpy or pyrr is **extremely slow** in the scope of real time rendering. Try to calculate matrixes ahead of time. Also moving the matrix calculations inside the shader programs can help greatly.

You can easily do 1000 draw calls of a small to medium vertex array and still run 60+ fps even on older hardware. The minute you throw in some matrix calculation in that loop you might be able to draw 50 before the framerate tanks.

This can also be solved by using more efficient libraries. miniglm. have been one suggestion that looks promising.

2.4.2 Conclusion

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.

ModernGL have tools like `Query` and `ConditionalRender` that can be very helpful in measuring and improving performance. See the ModernGL documentation for more info.

We can also strive to do more with less. Rendering, in the end, is really just about creating illusions. Still, it's quite amazing what can be achieved with OpenGL and Python today when using the right tools and methods.

2.5 Audio

The current music timers do a decent job reporting the current time, but more work needs to be done to find better alternative for accurate audio playback.

We separate playback libraries in two types based on their capabilities.

1. Accurate reporting of current time
2. Accurate reporting of current time and fast and accurate time seeking

These capabilites should also ideally work across the tree main platforms: Linux, OS X and Windows.

We have decent type 1 timers, but more work needs to be done to find better type 2 libraries. This is important when working with timing tools such as rocket and when jumping around in the timeline.

Some of the current timers also work inconsistenly between platforms. A lot more research and work is needed.

Note: Contributions in any form on this topic is greatly appreciated.

CHAPTER 3

Controls

3.1 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*)
- `R` reload shader programs (Needs configuration)
- `LEFT` jump 10 seconds back in time
- `RIGHT` jump 10 seconds forward in time

3.2 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

Mouse Controls

- Standard yaw/pitch camera rotation with mouse

CHAPTER 4

Reference

4.1 demosys.effects.Effect

demosys.effects.**Effect**

The base Effect base class that should be extended when making an effect.

The typical example:

```
import moderngl
from demosys.effects import Effect
from demosys import geometry

class MyEffect(Effect):
    def __init__(self):
        # Initalization happens after resources are loaded
        self.program = self.get_program("my_program_label")
        self.fullscreen_quad = geometry.quad_fs()

    def post_load(self):
        # Initialization after all effects are initialized

    def draw(self, time, frametime, target):
        # Render a colored fullscreen quad
        self.ctx.enable_only(moderngl.DEPTH_TEST)
        self.program["color"].value = (1.0, 1.0, 1.0, 1.0)
        self.fullscreen_quad.render(self.program)
```

4.1.1 Initialization

Effect.**__init__**(**args*, ***kwargs*)

Implement the initialize when extending the class. This method is responsible for fetching or creating resources and doing genereal initalization of the effect.

The effect initializer is called when all resources are loaded (with the exception of resources you manually load in the the initializer).

If your effect requires arguments during initialiation you are free to add positional and keyword arguments.

You **do not** have to call the superclass initializer though `super()`

Example:

```
def __init__(self):
    # Fetch reference to resource by their label
    self.program = self.get_program('simple_textured')
    self.texture = self.get_texture('bricks')
    # .. create a cube etc ..
```

`Effect.`**`post_load`**`()`
Called after all effects are initialized before drawing starts. Some initialization may be neccessary to do here such as interaction with other effects.

This method does nothing unless implemented.

4.1.2 Draw Methods

`Effect.`**`draw`**(*time: float*, *frametime: float*, *target: moderngl.framebuffer.Framebuffer*)
Draw function called by the system every frame when the effect is active. This method raises `NotImplementedError` unless implemented.

Parameters

- **`time`** (*`float`*) – The current time in seconds.
- **`frametime`** (*`float`*) – The time the previous frame used to render in seconds.
- **`target`** (`moderngl.Framebuffer`) – The target FBO for the effect.

4.1.3 Resource Methods

`Effect.`**`get_texture`**(*label: str*) → Union[moderngl.texture.Texture, moderngl.texture_array.TextureArray, moderngl.texture_3d.Texture3D, moderngl.texture_cube.TextureCube]
Get a texture by its label

Parameters **`label`** (*`str`*) – The Label for the texture

Returns The py:class:*moderngl.Texture* instance

`Effect.`**`get_program`**(*label: str*) → moderngl.program.Program
Get a program by its label

Parameters **`label`** (*`str`*) – The label for the program

Returns: py:class:*moderngl.Program* instance

`Effect.`**`get_scene`**(*label: str*) → demosys.scene.scene.Scene
Get a scene by its label

Parameters **`label`** (*`str`*) – The label for the scene

Returns: The `Scene` instance

`Effect.`**`get_data`**`()`
Get a data instance by its label

Parameters **label** (*str*) – Label for the data instance

Returns Contents of the data file

`Effect.`**`get_effect`**(*label: str*) → demosys.effects.effect.Effect
Get an effect instance by label.

Parameters **label** (*str*) – Label for the data file

Returns: The *Effect* instance

`Effect.`**`get_effect_class`**(*effect_name: str*, *package_name: str = None*) → Type[demosys.effects.effect.Effect]
Get an effect class by the class name

Parameters **effect_name** (*str*) – Name of the effect class

Keyword Arguments **package_name** (*str*) – The package the effect belongs to. This is optional and only needed when effect class names are not unique.

Returns *Effect* class

`Effect.`**`get_track`**(*name: str*) → rocket.tracks.Track
Gets or creates a rocket track. Only avaiable when using a Rocket timer.

Parameters **name** (*str*) – The rocket track name

Returns The `rocket.Track` instance

4.1.4 Utility Methods

`Effect.`**`create_projection`**(*fov: float = 75.0*, *near: float = 1.0*, *far: float = 100.0*, *aspect_ratio: float = None*)
Create a projection matrix with the following parameters. When `aspect_ratio` is not provided the configured aspect ratio for the window will be used.

Parameters

- **fov** (*float*) – Field of view (float)
- **near** (*float*) – Camera near value
- **far** (*float*) – Camrea far value

Keyword Arguments **aspect_ratio** (*float*) – Aspect ratio of the viewport

Returns The projection matrix as a float32 `numpy.array`

`Effect.`**`create_transformation`**(*rotation=None*, *translation=None*)
Creates a transformation matrix woth rotations and translation.

Parameters

- **rotation** – 3 component vector as a list, tuple, or `pyrr.Vector3`
- **translation** – 3 component vector as a list, tuple, or `pyrr.Vector3`

Returns A 4x4 matrix as a `numpy.array`

`Effect.`**`create_normal_matrix`**(*modelview*)
Creates a normal matrix from modelview matrix

Parameters **modelview** – The modelview matrix

Returns A 3x3 Normal matrix as a `numpy.array`

4.1.5 Attributes

`Effect.`**`runnable = True`**
The runnable status of the effect instance. A runnable effect should be able to run with the `runeffect` command or run in a project

`Effect.`**`ctx`**
The ModernGL context

`Effect.`**`window`**
The `Window`

`Effect.`**`sys_camera`**
The system camera responding to input

`Effect.`**`name`**
Full python path to the effect

`Effect.`**`label`**
The label assigned to this effect instance

4.2 demosys.project.base.BaseProject

`demosys.project.base.`**`BaseProject`**
The base project class we extend when creating a project configuration

The minimal implementation:

```
from demosys.project.base import BaseProject
from demosys.resources.meta import ProgramDescription, TextureDescription

class Project(BaseProject):
    # The effect packages to import using full python path
    effect_packages = [
        'myproject.efect_package1',
        'myproject.efect_package2',
        'myproject.efect_package2',
    ]
    # Resource description for global project resources (not loaded by effect␣
↪packages)
    resources = [
        ProgramDescription(label='cube_textured', path="cube_textured.glsl'),
        TextureDescription(label='wood', path="wood.png', mipmap=True),
    ]

    def create_resources(self):
        # Override the method adding additional resources

        # Create some shared fbo
        size = (256, 256)
        self.shared_framebuffer = self.ctx.framebuffer(
            color_attachments=self.ctx.texture(size, 4),
            depth_attachement=self.ctx.depth_texture(size)
        )

        return self.resources
```

(continues on next page)

(continued from previous page)

```
    def create_effect_instances(self):
        # Create and register instances of an effect class we loaded from the␣
↪effect packages
        self.create_effect('cube1', 'CubeEffect')

        # Using full path to class
        self.create_effect('cube2', 'myproject.efect_package1.CubeEffect')

        # Passing variables to initializer
        self.create_effect('cube3', 'CubeEffect', texture=self.get_texture('wood
↪'))

        # Assign resources manually
        cube = self.create_effect('cube1', 'CubeEffect')
        cube.program = self.get_program('cube_textured')
        cube.texture = self.get_texture('wood')
        cube.fbo = self.shared_framebuffer
```

These effects instances can then be obtained by the configured timeline class deciding when they should be rendered.

4.2.1 Create Methods

BaseProject.**create_effect**(*label: str*, *name: str*, **args*, ***kwargs*) → demosys.effects.effect.Effect
Create an effect instance adding it to the internal effects dictionary using the label as key.

Parameters

- **label** (*str*) – The unique label for the effect instance
- **name** (*str*) – Name or full python path to the effect class we want to instantiate
- **args** – Positional arguments to the effect initializer
- **kwargs** – Keyword arguments to the effect initializer

Returns The newly created Effect instance

BaseProject.**create_effect_classes**()
Registers effect packages defined in `effect_packages`.

BaseProject.**create_resources**() → List[demosys.resources.base.ResourceDescription]
Create resources for the project. Simply returns the `resources` list and can be implemented to modify what a resource list is programmatically.

Returns List of resource descriptions to load

BaseProject.**create_external_resources**() → List[demosys.resources.base.ResourceDescription]
Fetches all resource descriptions defined in effect packages.

Returns List of resource descriptions to load

BaseProject.**create_effect_instances**()
Create instances of effects. Must be implemented or `NotImplementedError` is raised.

4.2.2 Resource Methods

BaseProject.**get_effect**(*label: str*) → demosys.effects.effect.Effect
Get an effect instance by label

Parameters **label** (*str*) – The label for the effect instance

Returns Effect class instance

BaseProject.**get_effect_class**(*class_name*, *package_name=None*) → Type[demosys.effects.effect.Effect]
Get an effect class from the effect registry.

Parameters **class_name** (*str*) – The exact class name of the effect

Keyword Arguments **package_name** (*str*) – The python path to the effect package the effect name is located. This is optional and can be used to avoid issue with class name collisions.

Returns Effect class

BaseProject.**get_scene**(*label: str*) → demosys.scene.scene.Scene
Gets a scene by label

Parameters **label** (*str*) – The label for the scene to fetch

Returns Scene instance

BaseProject.**get_program**(*label: str*) → moderngl.program.Program

BaseProject.**get_texture**(*label: str*) → Union[moderngl.texture.Texture, moderngl.texture_array.TextureArray, moderngl.texture_3d.Texture3D, moderngl.texture_cube.TextureCube]
Get a texture by label

Parameters **label** (*str*) – The label for the texture to fetch

Returns Texture instance

BaseProject.**get_data**()
Get a data resource by label

Parameters **label** (*str*) – The labvel for the data resource to fetch

Returns The requeted data object

4.2.3 Other Methods

BaseProject.**load**()
Loads this project instance

BaseProject.**post_load**()
Called after resources are loaded before effects starts rendering. It simply iterates each effect instance calling their `post_load` methods.

BaseProject.**reload_programs**()
Reload all shader programs with the reloadable flag set

BaseProject.**get_runnable_effects**() → List[demosys.effects.effect.Effect]
Returns all runnable effects in the project.

Returns List of all runnable effects

4.2.4 Attributes

`BaseProject.`**`effect_packages = []`**
The effect packages to load

`BaseProject.`**`resources = []`**
Global project resource descriptions

`BaseProject.`**`ctx`**
The MondernGL context

4.3 demosys.opengl.vao.VAO

class `demosys.opengl.vao.`**VAO**(*name=''*, *mode=4*)
Represents a vertex array object. This is a wrapper class over `moderngl.VertexArray` to provide helper method.

The main purpose is to provide render methods taking a program as parameter. The class will auto detect the programs attributes and add padding when needed to match the vertex object. A new vertexbuffer object is created and stored internally for each unique shader program used.

A secondary purpose is to provide an alternate way to build vertexbuffers This can be practical when loading or creating various geometry.

There is no requirements to use this class, but most methods in the system creating vertexbuffers will return this type. You can obtain a single vertexbuffer instance by calling *`VAO.instance()`* method if you prefer to work directly on moderngl instances.

4.3.1 Create

`VAO.`**`__init__`**(*name=''*, *mode=4*)
Create and empty VAO

Keyword Arguments

- **`name`** (*`str`*) – The name for debug purposes
- **`mode`** (*`int`*) – Default draw mode

`VAO.`**`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 data. Can be `numpy.array`, `moderngl.Buffer` or `bytes`.
- **`buffer_format`** (*`str`*) – The format of the buffer. (eg. `3f 3f` for interleaved positions and normals).
- **`attribute_names`** – A list of attribute names this buffer should map to.

Keyword Arguments **`per_instance`** (*`bool`*) – Is this buffer per instance data for instanced rendering?

Returns The `moderngl.Buffer` instance object. This is handy when providing `bytes` and `numpy.array`.

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

Parameters **buffer** – moderngl.Buffer, numpy.array or bytes

Keyword Arguments **index_element_size** (*int*) – Byte size of each element. 1, 2 or 4

4.3.2 Render Methods

VAO.**render**(*program: moderngl.program.Program*, *mode=None*, *vertices=-1*, *first=0*, *instances=1*)
Render the VAO.

Parameters **program** – The moderngl.Program

Keyword Arguments

- **mode** – Override the draw mode (TRIANGLES etc)
- **vertices** (*int*) – The number of vertices to transform
- **first** (*int*) – The index of the first vertex to start with
- **instances** (*int*) – The number of instances

VAO.**render_indirect**(*program: moderngl.program.Program*, *buffer*, *mode=None*, *count=-1*, ***, *first=0*)
The render primitive (mode) must be the same as the input primitive of the GeometryShader. The draw commands are 5 integers: (count, instanceCount, firstIndex, baseVertex, baseInstance).

Parameters

- **program** – The moderngl.Program
- **buffer** – The moderngl.Buffer containing indirect draw commands

Keyword Arguments

- **mode** (*int*) – By default TRIANGLES will be used.
- **count** (*int*) – The number of draws.
- **first** (*int*) – The index of the first indirect draw command.

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

Parameters

- **program** – The moderngl.Program
- **buffer** – The moderngl.buffer to store the output

Keyword Arguments

- **mode** – Draw mode (for example moderngl.POINTS)
- **vertices** (*int*) – The number of vertices to transform
- **first** (*int*) – The index of the first vertex to start with
- **instances** (*int*) – The number of instances

4.3.3 Other Methods

VAO.**instance**(*program:Program*) → VertexArray
Obtain the moderngl.VertexArray instance for the program. The instance is only created once and cached internally.

Returns: moderngl.VertexArray instance

VAO.**release**(*buffer=True*)
Destroy the vao object

Keyword Arguments **buffers** (*bool*) – also release buffers

4.4 demosys.geometry

The geometry module is a collection of functions generating simple geometry / VAOs.

4.4.1 Functions

demosys.geometry.**quad_fs**() → demosys.opengl.vao.VAO
Creates a screen aligned quad using two triangles with normals and texture coordiantes.

Returns A *demosys.opengl.vao.VAO* instance.

demosys.geometry.**quad_2d**(*width*, *height*, *xpos=0.0*, *ypos=0.0*) → demosys.opengl.vao.VAO
Creates a 2D quad VAO using 2 triangles with normals and texture coordinates.

Parameters

- **width** (*float*) – Width of the quad
- **height** (*float*) – Height of the quad

Keyword Arguments

- **xpos** (*float*) – Center position x
- **ypos** (*float*) – Center position y

Returns A *demosys.opengl.vao.VAO* instance.

demosys.geometry.**cube**(*width*, *height*, *depth*, *center=(0.0, 0.0, 0.0)*, *normals=True*, *uvs=True*) → demosys.opengl.vao.VAO
Creates a cube VAO with normals and texture coordinates

Parameters

- **width** (*float*) – Width of the cube
- **height** (*float*) – Height of the cube
- **depth** (*float*) – Depth of the cube

Keyword Arguments

- **center** – center of the cube as a 3-component tuple
- **normals** – (bool) Include normals
- **uvs** – (bool) include uv coordinates

Returns A *demosys.opengl.vao.VAO* instance

demosys.geometry.**bbox**(*width=1.0*, *height=1.0*, *depth=1.0*)
Generates a bounding box with (0.0, 0.0, 0.0) as the center. This is simply a box with LINE_STRIP as draw mode.

Keyword Arguments

- **width** (*float*) – Width of the box
- **height** (*float*) – Height of the box
- **depth** (*float*) – Depth of the box

Returns A *demosys.opengl.vao.VAO* instance

demosys.geometry.**plane_xz**(*size=(10, 10)*, *resolution=(10, 10)*) → demosys.opengl.vao.VAO
Generates a plane on the xz axis of a specific size and resolution. Normals and texture coordinates are also included.

Parameters

- **size** – (x, y) tuple
- **resolution** – (x, y) tuple

Returns A *demosys.opengl.vao.VAO* instance

demosys.geometry.**points_random_3d**(*count*, *range_x=(-10.0, 10.0)*, *range_y=(-10.0, 10.0)*, *range_z=(-10.0, 10.0)*, *seed=None*) → demosys.opengl.vao.VAO
Generates random positions inside a confied box.

Parameters **count** (*int*) – Number of points to generate

Keyword Arguments

- **range_x** (*tuple*) – min-max range for x axis: Example (-10.0. 10.0)
- **range_y** (*tuple*) – min-max range for y axis: Example (-10.0. 10.0)
- **range_z** (*tuple*) – min-max range for z axis: Example (-10.0. 10.0)
- **seed** (*int*) – The random seed

Returns A *demosys.opengl.vao.VAO* instance

demosys.geometry.**sphere**(*radius=0.5*, *sectors=32*, *rings=16*) → demosys.opengl.vao.VAO
Creates a sphere.

Keyword Arguments

- **radius** (*float*) – Radius or the sphere
- **rings** (*int*) – number or horizontal rings
- **sectors** (*int*) – number of vertical segments

Returns A *demosys.opengl.vao.VAO* instance

4.5 demosys.timers.base.BaseTimer

demosys.timers.base.**BaseTimer = <class 'demosys.timers.base.BaseTimer'>**
The base class guiding the implementation of timers. All methods must be implemented.

4.5.1 Methods

`BaseTimer.`**`start`**`()`
Start the timer initially or resume after pause

Raises `NotImplementedError`

`BaseTimer.`**`stop`**`()` → float
Stop the timer. Should only be called once when stopping the timer.

Returns The time the timer was stopped

Raises `NotImplementedError`

`BaseTimer.`**`pause`**`()`
Pause the timer

Raises `NotImplementedError`

`BaseTimer.`**`toggle_pause`**`()`
Toggle pause state

Raises `NotImplementedError`

`BaseTimer.`**`get_time`**`()` → float
Get the current time in seconds

Returns The current time in seconds

Raises `NotImplementedError`

`BaseTimer.`**`set_time`**`(`*value: float*`)`
Set the current time in seconds.

Parameters **`value`** (*`float`*) – The new time

Raises `NotImplementedError`

4.6 demosys.timers.clock.Timer

`demosys.timers.clock.`**`Timer`**
Timer based on python `time`. This is the default timer.

4.6.1 Methods

`Timer.`**`start`**`()`
Start the timer by recoding the current `time.time()` preparing to report the number of seconds since this timestamp.

`Timer.`**`stop`**`()` → float
Stop the timer

Returns The time the timer was stopped

`Timer.`**`pause`**`()`
Pause the timer by setting the internal pause time using `time.time()`

`Timer.`**`toggle_pause`**`()`
Toggle the paused state

`Timer.`**`get_time`**`()` → float
Get the current time in seconds

Returns The current time in seconds

`Timer.`**`set_time`**`(`*value: float*`)`
Set the current time. This can be used to jump in the timeline.

Parameters **`value`** (*`float`*) – The new time

4.7 demosys.timers.music.Timer

`demosys.timers.music.`**`Timer`**
Timer based on the current position in a wav, ogg or mp3 using pygame.mixer. Path to the music file is configured in `settings.MUSIC`.

4.7.1 Methods

`Timer.`**`start`**`()`
Play the music

`Timer.`**`stop`**`()` → float
Stop the music

Returns The current location in the music

`Timer.`**`pause`**`()`
Pause the music

`Timer.`**`toggle_pause`**`()`
Toggle pause mode

`Timer.`**`get_time`**`()` → float
Get the current position in the music in seconds

`Timer.`**`set_time`**`(`*value: float*`)`
Set the current time in the music in seconds causing the player to seek to this location in the file.

4.8 demosys.timers.rocket.Timer

`demosys.timers.rocket.`**`Timer`**
Basic rocket timer. Sets up rocket using values in `settings.ROCKET`. The current time is translated internally in rocket to row positions based on the configured rows per second (RPS).

4.8.1 Methods

`Timer.`**`start`**`()`
Start the timer

`Timer.`**`stop`**`()` → float
Stop the timer

Returns The current time.

`Timer.`**`pause`**`()`
Pause the timer

`Timer.`**`toggle_pause`**`()`
Toggle pause mode

`Timer.`**`get_time`**`()` → float
Get the current time in seconds

Returns The current time in seconds

`Timer.`**`set_time`**`(`*value: float*`)`
Set the current time jumping in the timeline.

Parameters **`value`** (*`float`*) – The new time

4.9 demosys.timers.rocketmusic.Timer

`demosys.timers.rocketmusic.`**`Timer`**
Combines music.Timer and rocket.Timer

4.9.1 Methods

`Timer.`**`start`**`()`
Start the timer

`Timer.`**`stop`**`()` → float
Stop the timer

Returns The current time

`Timer.`**`pause`**`()`
Pause the timer

`Timer.`**`toggle_pause`**`()`
Toggle pause mode

`Timer.`**`get_time`**`()` → float
Get the current time in seconds

Returns The current time in seconds

`Timer.`**`set_time`**`(`*value: float*`)`
Set the current time jumping in the timeline

Parameters **`value`** (*`float`*) – The new time value

4.10 demosys.timers.vlc.Timer

`demosys.timers.vlc.`**`Timer`**
Timer based on the python-vlc wrapper. Plays the music file defined in `settings.MUSIC`. Requires `python-vlc` to be installed including the vlc application.

4.10.1 Methods

`Timer.`**`start`**`()`
Start the music

`Timer.`**`stop`**`()` → float
Stop the music

Returns The current time in seconds

`Timer.`**`pause`**`()`
Pause the music

`Timer.`**`toggle_pause`**`()`
Toggle pause mode

`Timer.`**`get_time`**`()` → float
Get the current time in seconds

Returns The current time in seconds

`Timer.`**`set_time`**`(`*value: float*`)`
Set the current time in seconds.

Parameters **`value`** (*`float`*) – The new time

Raises `NotImplementedError`

4.11 demosys.context.base.BaseWindow

`demosys.context.base.`**`BaseWindow`**
The base window we extend when adding new window types to the system.

`demosys.context.base.`**`BaseKeys`**
Namespace for generic key constants working across all window types.

4.11.1 Methods

`BaseWindow.`**`__init__`**`()`
Base window intializer reading values from `settings`.

When creating the initializer in your own window always call this methods using `super().__init__()`.

The main responsebility of the initializer is to:

- initialize the window library
- identify the window framebuffer
- set up keyboard and mouse events
- create the `moderngl.Context` instance
- register the window in `context.WINDOW`

`BaseWindow.`**`draw`**`(`*current_time*, *frame_time*`)`
Draws a frame. Internally it calls the configured timeline's draw method.

Parameters

- **current_time** (*float*) – The current time (preferrably always from the configured timer class)
- **frame_time** (*float*) – The duration of the previous frame in seconds

BaseWindow.**clear**()
Clear the window buffer

BaseWindow.**clear_values**(*red=0.0*, *green=0.0*, *blue=0.0*, *alpha=0.0*, *depth=1.0*)
Sets the clear values for the window buffer.

Parameters

- **red** (*float*) – red compoent
- **green** (*float*) – green compoent
- **blue** (*float*) – blue compoent
- **alpha** (*float*) – alpha compoent
- **depth** (*float*) – depth value

BaseWindow.**use**()
Set the window buffer as the current render target

Raises NotImplementedError

BaseWindow.**swap_buffers**()
Swap the buffers. Most windows have at least support for double buffering cycling a back and front buffer.

Raises NotImplementedError

BaseWindow.**resize**(*width*, *height*)
Resize the window. Should normallty be overriden when implementing a window as most window libraries need additional logic here.

Parameters

- **width** (*int*) – Width of the window
- **height** – (int): Height of the window

BaseWindow.**close**()
Set the window in close state. This doesn't actually close the window, but should make *should_close()* return True so the main loop can exit gracefully.

Raises NotImplementedError

BaseWindow.**should_close**() → bool
Check if window should close. This should always be checked in the main draw loop.

Raises NotImplementedError

BaseWindow.**terminate**()
The actual teardown of the window.

Raises NotImplementedError

BaseWindow.**keyboard_event**(*key*, *action*, *modifier*)
Handles the standard keyboard events such as camera movements, taking a screenshot, closing the window etc.

Can be overriden add new keyboard events. Ensure this method is also called if you want to keep the standard features.

Parameters

- **key** – The key that was pressed or released
- **action** – The key action. Can be *ACTION_PRESS* or *ACTION_RELEASE*
- **modifier** – Modifiers such as holding shift or ctrl

BaseWindow.**cursor_event**(*x*, *y*, *dx*, *dy*)
The standard mouse movement event method. Can be overriden to add new functionality. By default this feeds the system camera with new values.

Parameters

- **x** – The current mouse x position
- **y** – The current mouse y position
- **dx** – Delta x postion (x position difference from the previous event)
- **dy** – Delta y postion (y position difference from the previous event)

BaseWindow.**print_context_info**()
Prints moderngl context info.

BaseWindow.**set_default_viewport**()
Calculates the viewport based on the configured aspect ratio in settings. Will add black borders if the window do not match the viewport.

4.11.2 Attributes

BaseWindow.**size**
(width, height) tuple containing the window size.

Note that for certain displays we rely on *buffer_size()* to get the actual window buffer size. This is fairly common for retina and 4k displays where the UI scale is > 1.0

BaseWindow.**buffer_size**
(width, heigh) buffer size of the window.

This is the actual buffer size of the window taking UI scale into account. A 1920 x 1080 window running in an environment with UI scale 2.0 would have a 3840 x 2160 window buffer.

BaseWindow.**keys = None**
The key class/namespace used by the window defining keyboard constants

4.12 demosys.context.pyqt.Window

demosys.context.pyqt.**Window**
Window using PyQt5.

This is the recommended window if you want your project to work on most platforms out of the box without any binary dependecies.

demosys.context.pyqt.**Keys**
Namespace creating pyqt specific key constants

4.12.1 Methods

`Window.__init__()`
Creates a pyqt application and window overriding the built in event loop. Sets up keyboard and mouse events and creates a `monderngl.Context`.

`Window.keyPressEvent(event)`
Pyqt specific key press callback function. Translates and forwards events to `keyboard_event()`.

`Window.keyReleaseEvent(event)`
Pyqt specific key release callback function. Translates and forwards events to `keyboard_event()`.

`Window.mouseMoveEvent(event)`
Pyqt specific mouse event callback Translates and forwards events to `cursor_event()`.

`Window.swap_buffers()`
Swaps buffers, increments the frame counter and pulls events

`Window.use()`
Make the window's framebuffer the current render target

`Window.should_close()` → bool
Checks if the internal close state is set

`Window.close()`
Set the internal close state

`Window.terminate()`
Quits the running qt application

4.12.2 Attributes

4.13 demosys.context.glfw.Window

`demosys.context.glfw.Window`
Window implementation using pyGLFW

`demosys.context.glfw.Keys`
Namespace defining glfw specific keys constants

4.13.1 Methods

`Window.__init__()`
Initializes glfw, sets up key and mouse events and creates a `moderngl.Context` using the context glfw createad.

Using the glfw window requires glfw binaries and pyGLFW.

`Window.use()`
Bind the window framebuffer making it the current render target

`Window.swap_buffers()`
Swaps buffers, incement the framecounter and pull events.

`Window.resize(width, height)`
Sets the new size and buffer size internally

Window.**close**()
Set the window closing state in glfw

Window.**should_close**()
Ask glfw is the window should be closed

Window.**terminate**()
Terminates the glfw library

Window.**key_event_callback**(*window*, *key*, *scancode*, *action*, *mods*)
Key event callback for glfw. Translates and forwards keyboard event to *keyboard_event()*

Parameters

- **window** – Window event origin
- **key** – The key that was pressed or released.
- **scancode** – The system-specific scancode of the key.
- **action** – GLFW_PRESS, GLFW_RELEASE or GLFW_REPEAT
- **mods** – Bit field describing which modifier keys were held down.

Window.**mouse_event_callback**(*window*, *xpos*, *ypos*)
Mouse event callback from glfw. Translates the events forwarding them to *cursor_event()*.

Parameters

- **window** – The window
- **xpos** – viewport x pos
- **ypos** – viewport y pos

Window.**window_resize_callback**(*window*, *width*, *height*)
Window resize callback for glfw

Parameters

- **window** – The window
- **width** – New width
- **height** – New height

Window.**poll_events**()
Poll events from glfw

Window.**check_glfw_version**()
Ensure glfw library version is compatible

4.13.2 Other Inherited Methods

Window.**draw**(*current_time*, *frame_time*)
Draws a frame. Internally it calls the configured timeline's draw method.

Parameters

- **current_time** (*float*) – The current time (preferrably always from the configured timer class)
- **frame_time** (*float*) – The duration of the previous frame in seconds

Window.**clear**()
Clear the window buffer

Window.**clear_values**(*red=0.0*, *green=0.0*, *blue=0.0*, *alpha=0.0*, *depth=1.0*)
Sets the clear values for the window buffer.

Parameters

- **red** (*float*) – red compoent
- **green** (*float*) – green compoent
- **blue** (*float*) – blue compoent
- **alpha** (*float*) – alpha compoent
- **depth** (*float*) – depth value

Window.**keyboard_event**(*key*, *action*, *modifier*)
Handles the standard keyboard events such as camera movements, taking a screenshot, closing the window etc.

Can be overriden add new keyboard events. Ensure this method is also called if you want to keep the standard features.

Parameters

- **key** – The key that was pressed or released
- **action** – The key action. Can be *ACTION_PRESS* or *ACTION_RELEASE*
- **modifier** – Modifiers such as holding shift or ctrl

Window.**cursor_event**(*x*, *y*, *dx*, *dy*)
The standard mouse movement event method. Can be overriden to add new functionality. By default this feeds the system camera with new values.

Parameters

- **x** – The current mouse x position
- **y** – The current mouse y position
- **dx** – Delta x postion (x position difference from the previous event)
- **dy** – Delta y postion (y position difference from the previous event)

Window.**print_context_info**()
Prints moderngl context info.

Window.**set_default_viewport**()
Calculates the viewport based on the configured aspect ratio in settings. Will add black borders if the window do not match the viewport.

4.13.3 Attributes

Window.**size**
(width, height) tuple containing the window size.

Note that for certain displays we rely on *buffer_size()* to get the actual window buffer size. This is fairly common for retina and 4k displays where the UI scale is > 1.0

Window.**buffer_size**
(width, heigh) buffer size of the window.

This is the actual buffer size of the window taking UI scale into account. A 1920 x 1080 window running in an environment with UI scale 2.0 would have a 3840 x 2160 window buffer.

Window.**keys = <class 'demosys.context.glfw.keys.Keys'>**

`Window.`**`min_glfw_version = (3, 2, 1)`**
The minimum glfw version required

4.14 demosys.context.headless.Window

`demosys.context.headless.`**`Window`**
Headless window using a standalone `moderngl.Context`.

4.14.1 Methods

`Window.`**`__init__`**`()`
Creates a standalone `moderngl.Context`. The headless window currently have no event input from keyboard or mouse.

Using this window require either `settings` values to be present:

- `HEADLESS_FRAMES`: How many frames should be rendered before closing the window
- `HEADLESS_DURATION`: How many seconds rendering should last before the window closes

`Window.`**`draw`**`(`*current_time*, *frame_time*`)`
Calls the superclass `draw()` methods and checks `HEADLESS_FRAMES`/`HEADLESS_DURATION`

`Window.`**`use`**`()`
Binds the framebuffer representing this window

`Window.`**`should_close`**`()` → bool
Checks if the internal close state is set

`Window.`**`close`**`()`
Sets the internal close state

`Window.`**`resize`**`(`*width*, *height*`)`
Resizing is not supported by the headless window. We simply override with an empty method.

`Window.`**`swap_buffers`**`()`
Headless window currently don't support double buffering. We only increment the frame counter here.

`Window.`**`terminate`**`()`
No teardown is needed. We override with an empty method

4.14.2 Other Inherited Methods

`Window.`**`set_default_viewport`**`()`
Calculates the viewport based on the configured aspect ratio in settings. Will add black borders if the window do not match the viewport.

`Window.`**`cursor_event`** (*x*, *y*, *dx*, *dy*)
The standard mouse movement event method. Can be overriden to add new functionality. By default this feeds the system camera with new values.

Parameters

- **`x`** – The current mouse x position
- **`y`** – The current mouse y position
- **`dx`** – Delta x postion (x position difference from the previous event)

- **dy** – Delta y postion (y position difference from the previous event)

Window.**keyboard_event**(*key*, *action*, *modifier*)
Handles the standard keyboard events such as camera movements, taking a screenshot, closing the window etc.

Can be overriden add new keyboard events. Ensure this method is also called if you want to keep the standard features.

Parameters

- **key** – The key that was pressed or released
- **action** – The key action. Can be *ACTION_PRESS* or *ACTION_RELEASE*
- **modifier** – Modifiers such as holding shift or ctrl

Window.**clear**()
Clear the window buffer

Window.**clear_values**(*red=0.0*, *green=0.0*, *blue=0.0*, *alpha=0.0*, *depth=1.0*)
Sets the clear values for the window buffer.

Parameters

- **red** (*float*) – red compoent
- **green** (*float*) – green compoent
- **blue** (*float*) – blue compoent
- **alpha** (*float*) – alpha compoent
- **depth** (*float*) – depth value

Window.**print_context_info**()
Prints moderngl context info.

4.14.3 Attributes

Window.**size**
(width, height) tuple containing the window size.

Note that for certain displays we rely on *buffer_size()* to get the actual window buffer size. This is fairly common for retina and 4k displays where the UI scale is > 1.0

Window.**buffer_size**
(width, heigh) buffer size of the window.

This is the actual buffer size of the window taking UI scale into account. A 1920 x 1080 window running in an environment with UI scale 2.0 would have a 3840 x 2160 window buffer.

Window.**keys = None**

4.15 demosys.context.pyglet.Window

demosys.context.pyglet.**Window**
Window based on pyglet.

Note that pylget is unable to make core 3.3+ contexts and will not work for certain drivers and enviroments such as on OS X.

demosys.context.pyglet.**Keys**
Namespace mapping pyglet specific key constants

4.15.1 Methods

Window.**__init__**()
Opens a window using pyglet, registers input callbacks and creates a moderngl context.

Window.**on_key_press**(*symbol*, *modifiers*)
Pyglet specific key press callback. Forwards and translates the events to keyboard_event()

Window.**on_key_release**(*symbol*, *modifiers*)
Pyglet specific key release callback. Forwards and translates the events to keyboard_event()

Window.**on_mouse_motion**(*x*, *y*, *dx*, *dy*)
Pyglet specific mouse motion callback. Forwards and traslates the event to cursor_event()

Window.**on_resize**(*width*, *height*)
Pyglet specific callback for window resize events.

Window.**use**()
Render to this window

Window.**swap_buffers**()
Swap buffers, increment frame counter and pull events

Window.**should_close**() → bool
returns the has_exit state in the pyglet window

Window.**close**()
Sets the close state in the pyglet window

Window.**terminate**()
No cleanup is really needed. Empty method

4.15.2 Attributes

CHAPTER 5

Settings

The `settings.py` file must be present in your project in order to run the framework.

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.

5.1 OPENGL

Sets the minimum required OpenGL version to run your project. A forward compatible core context will be always be requested. 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.

Note: To make your project work on OS X you cannot move past version 4.1.

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

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

5.2 WINDOW

Window/screen properties. Most importantly the `class` attribute decides what class should be used to handle the window.

The currently supported classes are:

- `demosys.context.pyqt.Window` PyQt5 window (default)

- `demosys.context.glfw.Window` pyGLFW window
- `demosys.context.pyglet.Window` Pyglet window (Not for OS X)
- `demosys.context.headless.Window` Headless window

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

Other Properties:

- `size`: The window size to open.
- `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.
- `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

5.3 SCREENSHOT_PATH

Absolute path to the directory screenshots will be saved. Screenshots will end up in the project root of not defined. If a path is configured, the directory will be auto-created.

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

5.4 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 /guides/audio section.

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

5.5 TIMER

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

```
TIMER = 'demosys.timers.clock.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.

Custom timers can be created. More information can be found in the /user_guide/timers section.

5.6 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,
}
```

5.7 TIMELINE

A timeline is a class deciding what effect(s) should be rendered (including order) at any given point in time.

```
# Default timeline only rendeing a single effect at all times
TIMELINE = 'demosys.timeline.single.Timeline'
```

You can create your own class handling this logic. More info in the /user_guide/timeline section.

5.8 PROGRAM_DIRS/PROGRAM_FINDERS

`PROGRAM_DIRS` contains absolute paths the `FileSystemFinder` will look for shaders programs.

`EffectDirectoriesFinder` will look for programs in all registered effect packages in the order they were added. This assumes you have a `resources/programs` directory in your effect packages.

A resource can have the same path in multiple locations. The system will return the last occurance of the resource. This way it is possible to override resources.

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

# Register a project-global programs directory
# These paths are searched last
PROGRAM_DIRS = (
    os.path.join(PROJECT_DIR, 'resources/programs'),
)
```

`PROGRAM_DIRS` can really be any directory and doesn't need to end with `/programs`

5.9 PROGRAM_LOADERS

Program loaders are classes responsible for loading resources. Custom loaders can easily be created.

Programs have a default set of loaders if not specified.

```
PROGRAM_LOADERS = (
    'demosys.loaders.program.single.Loader',
    'demosys.loaders.program.separate.Loader',
)
```

5.10 TEXTURE_DIRS/TEXTURE_FINDERS

Same principle as `PROGRAM`_DIRS and `PROGRAM_FINDERS`. The `EffectDirectoriesFinder` will look for a `textures` directory in effects.

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

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

5.11 TEXTURE_LOADERS

Texture loaders are classes responsible for loading textures. These can be easily customized.

The default texture loaders:

```
TEXTURE_LOADERS = (
    'demosys.loaders.texture.t2d.Loader',
    'demosys.loaders.texture.array.Loader',
)
```

5.12 SCENE_DIRS/SCENE_FINDERS

Same principle as `PROGRAM_DIRS` and `PROGRAM_FINDERS`. This is where scene files such as wavefront and gltf files are loaded from. The `EffectDirectoriesFinder` will look for a `scenes` directory

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

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

5.13 SCENE_LOADERS

Scene loaders are classes responsible for loading scenes or geometry from different formats.

The default scene loaders are:

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

5.14 DATA_DIRS/DATA_FINDERS

Same principle as `PROGRAM_DIRS` and `PROGRAM_FINDERS`. This is where the system looks for data files. These are generic loaders for binary, text and json data (or anything you want).

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

# Absolute path to a project-global scene directory
```

(continues on next page)

(continued from previous page)

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

5.15 DATA_LOADERS

Data loaders are classes responsible for loading miscellaneous data files. These are fairly easy to implement if you need to support something custom.

The default data loaders are:

```
DATA_LOADERS = (
    'demosys.loaders.data.binary.Loader',
    'demosys.loaders.data.text.Loader',
    'demosys.loaders.data.json.Loader',
)
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

B

C

D

Q

R

S

T

U

V

W