demosys-py Documentation

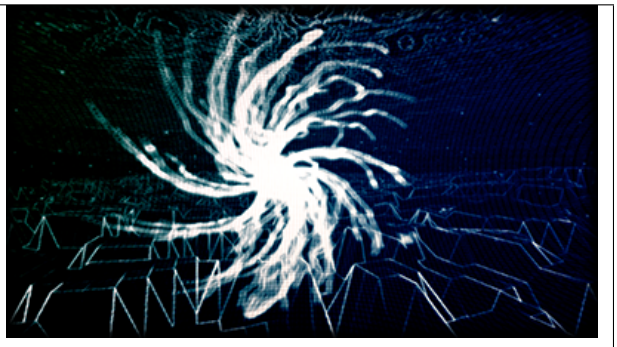
Release 2.0.0

Einar Forselv

Aug 21, 2018

Getting Started

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

Warning: The documentation for 2.0 is currently work in progress and you may find some sections outdated

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.

2.1.1 The Effect Package

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

```
cube
├── effects.py
├── dependencies.py
└── resources
    └── programs
        └── cube
            └── cube.glsl
    └── textures
    └── 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 the effect specific resources in the effect package they belong to instead of putting all resources for the entire project in the same location.

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

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 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

2.2.1 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`.

2.2.2 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>
```

2.2.3 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.

2.2.4 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")

    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.

2.3 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.

2.3.1 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',
)
```

2.3.2 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
        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 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`.

2.4.1 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

2.4.2 Custom Timer

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

class demosys.timers.base.**BaseTimer**(***kwargs*)
Timer based on glfw time

get_time()
Get the current time in seconds (float)

pause()
Pause the timer

set_time(*value*)
Set the current time

start()
Start the timer

stop()
Stop the timer

toggle_pause()
Toggle pause

2.5 Timeline

Timeline info

2.6 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.

2.6.1 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)
```

2.6.2 Type conversion

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

2.7 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 when possible.

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.

When doing many draw calls, do as little as possible between those draw calls. Doing matrix math in python even with numpy or pyrr is **extremely slow**. Try to calculate them ahead of time. Also moving the matrix calculations inside the shader programs can help greatly. You can easily do 1000 draw calls using the same cube 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.

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

2.8 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 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*)

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
- `R` reload shader programs (Needs configuration)

Mouse Controls

- Standard yaw/pitch camera rotation with mouse

CHAPTER 4

Reference

4.1 Effect

demosys.effects.**Effect**
Effect base class that should be extended when making an effect

Example:

```
from demosys.effects import Effect

class MyEffect(Effect):
    def __init__(self):
        # Initalization

    def draw(self, time, frametime, target):
        # Draw stuff
```

4.1.1 Initialization

Effect.**__init__**(**args*, ***kwargs*)
Implement the initialize when extending the class. This method is responsible for fetching resources and doing genereal initalization of the effect.

The effect initializer is called when all resources are loaded with the exception of resources loaded by other effects in the initializer.

The siganture of this method is entirely up to you.

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`**`()`
Override this method if needed when creating an effect.

Called after all effects are initialized before drawing starts. Some initialization may be neccessary to do here such as interaction with other effects.

4.1.2 Draw Methods

`Effect.`**`draw`**`(`*time*, *frametime*, *target*`)`
Draw function called by the system every frame when the effect is active. You are supposed to override this method.

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*`)` → moderngl.texture.Texture
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*`)` → 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*`)` → 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`**`(`*label*`)` → Any
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*`)` → demosys.effects.effect.Effect
Get an effect instance by label. This is only possible when you have your own Project

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

Returns: The *`Effect`* instance

Effect.**get_effect_class**(*effect_name*, *package_name=None*) → Type[Effect]
Get an effect class.

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

Keyword Arguments **package_name** (*str*) – The package the effect belongs to

Returns *Effect* class

Effect.**get_track**(*name*) → rocket.tracks.Track
This is only avaiable when using a Rocket timer.

Get or create a rocket track. This only makes sense when using rocket timers. If the resource is not loaded yet, an empty track object is returned that will be populated later.

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

Returns The `rocket.Track` instance

4.1.4 Utility Methods

Effect.**create_projection**(*fov=75.0*, *near=1.0*, *far=100.0*, *aspect_ratio=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.

:param : param float fov: Field of view (float) :param : param float near: Camera near value :param : param float far: Camrea far value

Parameters **ratio** (*float*) – Aspect ratio of the window

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`**
Full python path to the effect

4.2 VAO

`class` `demosys.opengl.vao.`**`VAO`**(*name*, *mode=4*)
Represents a vertex array object. A name must be provided for debug puporses. The default draw mode is `moderngl.TRIANGLES`

4.2.1 Methods

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

Returns The buffer object

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

Parameters

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

`VAO.`**`instance`**(*program:Program*) → VertexArray
Obtain the moderngl.VertexArray instance for the program

Returns moderngl.VertexArray

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

Parameters

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

`VAO.`**`render_indirect`**(*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** – (Buffer) Indirect drawing commands.
- **buffer** – (Buffer) Indirect drawing commands.
- **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:Program*, *buffer:Buffer*, *mode=None*, *vertices=-1*, *first=0*, *instances=1*)
Transform vertices. Stores the output in a single buffer.

Parameters

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

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

Parameters **buffers** – (bool) also release buffers

4.3 geometry

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

4.3.1 Functions

`demosys.geometry.quad_fs`() → VAO
Creates a screen aligned quad.

`demosys.geometry.quad_2d`(*width*, *height*, *xpos=0.0*, *ypos=0.0*) → VAO
Creates a 2D quad VAO using 2 triangles.

Parameters

- **width** – Width of the quad
- **height** – Height of the quad
- **xpos** – Center position x
- **ypos** – Center position y

`demosys.geometry.cube`(*width*, *height*, *depth*, *center=(0.0, 0.0, 0.0)*, *normals=True*, *uvs=True*) → VAO
Generates a cube VAO. The cube center is (0.0, 0.0 0.0) unless other is specified.

Parameters

- **width** – Width of the cube

- **height** – height of the cube
- **depth** – depth of the cube
- **center** – center of the cube
- **normals** – (bool) Include normals
- **uvs** – (bool) include uv coordinates

Returns VAO representing the cube

demosys.geometry.**bbox**(*width=1.0*, *height=1.0*, *depth=1.0*) → VAO

Generates a bounding box. This is simply a box with LINE_STRIP as draw mode

Parameters

- **width** – Width of the box
- **height** – height of the box
- **depth** – depth of the box

Returns VAO

demosys.geometry.**plane_xz**(*size=(10, 10)*, *resolution=(10, 10)*) → VAO

Generates a plane on the xz axis of a specific size and resolution

Parameters

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

Returns VAO

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*) → VAO

Generates random positions

Parameters

- **count** – Number of points
- **range_x** – min-max range for x axis
- **range_y** – min-max range for y axis
- **range_z** – min-max range for z axis
- **seed** – The random seed to be used

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

Generate a sphere

Parameters

- **radius** – Radius or the sphere
- **rings** – number or horizontal rings
- **sectors** – number of vertical segments

Returns VAO containing the sphere

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 *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 *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

E

G

I

L

N

P

Q

R

S

T

V

W