Skip to content

GPU Module (gpu)

This module provides Python wrappers for the GPU implementation in Blender. Some higher level functions can be found in the `gpu_extras` modul

SUBMODULES

GPU Types (gpu.types)
GPU Matrix Utilities (gpu.matrix)
GPU Select Utilities (gpu.select)
GPU Shader Utilities (gpu.shader)
GPU State Utilities (gpu.state)
GPU Texture Utilities (gpu.texture)
GPU Platform Utilities (gpu.platform)
GPU Capabilities Utilities (gpu.capabilities)

Geometry Batches

Geometry is drawn in batches. A batch contains the necessary data to perform the drawing. That includes an obligatory *Vertex Buffer* and an optional *Index Buffer*, each of which is described in more detail in the following sections. A batch also defines a draw type. Typical draw types are `POINTS`, `LINES` and `TRIS`. The draw type determines how the data will be interpreted and drawn.

Vertex Buffers

A *Vertex Buffer Object* (VBO) (`gpu.types.GPUVertBuf`) is an array that contains the vertex attributes needed for drawing using a specific shader. Typical vertex attributes are *location*, *normal*, *color*, and *uv*. Every vertex buffer has a *Vertex Format* (`gpu.types.GPUVertFormat`) and a length corresponding to the number of vertices in the buffer. A vertex format describes the attributes stored per vertex and their types.

The following code demonstrates the creation of a vertex buffer that contains 6 vertices. For each vertex 2 attributes will be stored: The position and the normal.

```
import gpu
vertex_positions = [(0, 0, 0), ...]
vertex_normals = [(0, 0, 1), ...]

fmt = gpu.types.GPUVertFormat()
fmt.attr_add(id="pos", comp_type='F32', len=3, fetch_mode='FLOAT')
fmt.attr_add(id="normal", comp_type='F32', len=3, fetch_mode='FLOAT')

vbo = gpu.types.GPUVertBuf(len=6, format=fmt)
vbo.attr_fill(id="pos", data=vertex_positions)
vbo.attr_fill(id="normal", data=vertex_normals)
```

This vertex buffer could be used to draw 6 points, 3 separate lines, 5 consecutive lines, 2 separate triangles, … E.g. in the case of lines, each two consecutive vertices define a line. The type that will actually be drawn is determined when the batch is created later.

Index Buffers

Often triangles and lines share one or more vertices. With only a vertex buffer one would have to store all attributes for the these vertices multiple times. This is very inefficient because in a connected triangle mesh every vertex is used 6 times on average. A more efficient approach would be to use an *Index Buffer* (IBO) (`gpu.types.GPUIndexBuf`), sometimes referred to as *Element Buffer*. An *Index Buffer* is an array that references vertices bas on their index in the vertex buffer.

For instance, to draw a rectangle composed of two triangles, one could use an index buffer.

```
positions = (
```

```
    (-1,  1), (1,  1),
    (-1, -1), (1, -1))

indices = ((0, 1, 2), (2, 1, 3))

ibo = gpu.types.GPUIndexBuf(type='TRIS', seq=indices)
```

Here the first tuple in `indices` describes which vertices should be used for the first triangle (same for the second tuple). Note how the diagonal vertic 1 and 2 are shared between both triangles.

Shaders

A shader is a program that runs on the GPU (written in GLSL in our case). There are multiple types of shaders. The most important ones are *Vertex Shaders* and *Fragment Shaders*. Typically multiple shaders are linked together into a *Program*. However, in the Blender Python API the term *Shader* refers to an OpenGL Program. Every `gpu.types.GPUShader` consists of a vertex shader, a fragment shader and an optional geometry shader. For common drawing tasks there are some built-in shaders accessible from `gpu.shader.from_builtin` with an identifier such as `UNIFORM_COLOR` or `FLAT_COLOR`.

Every shader defines a set of attributes and uniforms that have to be set in order to use the shader. Attributes are properties that are set using a vertex buffer and can be different for individual vertices. Uniforms are properties that are constant per draw call. They can be set using the `shader.uniform_*` functions after the shader has been bound.

Batch Creation

Batches can be creates by first manually creating VBOs and IBOs. However, it is recommended to use the `gpu_extras.batch.batch_for_shader` function. It makes sure that all the vertex attributes necessary for a specific shader are provided. Consequently, the shader has to be passed to the function as well. When using this function one rarely has to care about the vertex format, VBOs and IBOs created in the background. This is still something one should know when drawing stuff though.

Since batches can be drawn multiple times, they should be cached and reused whenever possible.

Offscreen Rendering

What one can see on the screen after rendering is called the *Front Buffer*. When draw calls are issued, batches are drawn on a *Back Buffer* that will on be displayed when all drawing is done and the current back buffer will become the new front buffer. Sometimes, one might want to draw the batches into distinct buffer that could be used as texture to display on another object or to be saved as image on disk. This is called Offscreen Rendering. In Blender Offscreen Rendering is done using the `gpu.types.GPUOffScreen` type.

Warning

`gpu.types.GPUOffScreen` objects are bound to the OpenGL context they have been created in. This means that once Blender discards this context (i.e. the window is closed), the offscreen instance will be freed.

Examples

To try these examples, just copy them into Blenders text editor and execute them. To keep the examples relatively small, they just register a draw functio that can't easily be removed anymore. Blender has to be restarted in order to delete the draw handlers.

3D Lines with Single Color

```
import bpy
import gpu
from gpu_extras.batch import batch_for_shader

coords = [(1, 1, 1), (-2, 0, 0), (-2, -1, 3), (0, 1, 1)]
shader = gpu.shader.from_builtin('UNIFORM_COLOR')
batch = batch_for_shader(shader, 'LINES', {"pos": coords})
```

```
batch = batch_for_shader(shader, 'LINES', {"pos": coords})


def draw():
    shader.uniform_float("color", (1, 1, 0, 1))
    batch.draw(shader)


bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_VIEW')
```

Triangle with Custom Shader

```
import bpy
import gpu
from gpu_extras.batch import batch_for_shader


vert_out = gpu.types.GPUStageInterfaceInfo("my_interface")
vert_out.smooth('VEC3', "pos")

shader_info = gpu.types.GPUShaderCreateInfo()
shader_info.push_constant('MAT4', "viewProjectionMatrix")
shader_info.push_constant('FLOAT', "brightness")
shader_info.vertex_in(0, 'VEC3', "position")
shader_info.vertex_out(vert_out)
shader_info.fragment_out(0, 'VEC4', "FragColor")

shader_info.vertex_source(
    "void main()"
    "{"
    "  pos = position;"
    "  gl_Position = viewProjectionMatrix * vec4(position, 1.0f);"
    "}"
)

shader_info.fragment_source(
    "void main()"
    "{"
    "  FragColor = vec4(pos * brightness, 1.0);"
    "}"
)

shader = gpu.shader.create_from_info(shader_info)
del vert_out
del shader_info

coords = [(1, 1, 1), (2, 0, 0), (-2, -1, 3)]
batch = batch_for_shader(shader, 'TRIS', {"position": coords})


def draw():
    matrix = bpy.context.region_data.perspective_matrix
    shader.uniform_float("viewProjectionMatrix", matrix)
    shader.uniform_float("brightness", 0.5)
    batch.draw(shader)
```

```
bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_VIEW')
```

Wireframe Cube using Index Buffer

```
import bpy
import gpu
from gpu_extras.batch import batch_for_shader

coords = (
    (-1, -1, -1), (+1, -1, -1),
    (-1, +1, -1), (+1, +1, -1),
    (-1, -1, +1), (+1, -1, +1),
    (-1, +1, +1), (+1, +1, +1))

indices = (
    (0, 1), (0, 2), (1, 3), (2, 3),
    (4, 5), (4, 6), (5, 7), (6, 7),
    (0, 4), (1, 5), (2, 6), (3, 7))

shader = gpu.shader.from_builtin('UNIFORM_COLOR')
batch = batch_for_shader(shader, 'LINES', {"pos": coords}, indices=indices)


def draw():
    shader.uniform_float("color", (1, 0, 0, 1))
    batch.draw(shader)


bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_VIEW')
```

Mesh with Random Vertex Colors

```
import bpy
import gpu
import numpy as np
from random import random
from gpu_extras.batch import batch_for_shader

mesh = bpy.context.active_object.data
mesh.calc_loop_triangles()

vertices = np.empty((len(mesh.vertices), 3), 'f')
indices = np.empty((len(mesh.loop_triangles), 3), 'i')

mesh.vertices.foreach_get(
    "co", np.reshape(vertices, len(mesh.vertices) * 3))
mesh.loop_triangles.foreach_get(
    "vertices", np.reshape(indices, len(mesh.loop_triangles) * 3))

vertex_colors = [(random(), random(), random(), 1) for _ in range(len(mesh.vertices))]

shader = gpu.shader.from_builtin('SMOOTH_COLOR')
```

```
shader = gpu.shader.from_builtin('SMOOTH_COLOR')
batch = batch_for_shader(
    shader, 'TRIS',
    {"pos": vertices, "color": vertex_colors},
    indices=indices,
)


def draw():
    gpu.state.depth_test_set('LESS_EQUAL')
    gpu.state.depth_mask_set(True)
    batch.draw(shader)
    gpu.state.depth_mask_set(False)


bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_VIEW')
```

2D Rectangle

```
import bpy
import gpu
from gpu_extras.batch import batch_for_shader

vertices = (
    (100, 100), (300, 100),
    (100, 200), (300, 200))

indices = (
    (0, 1, 2), (2, 1, 3))

shader = gpu.shader.from_builtin('UNIFORM_COLOR')
batch = batch_for_shader(shader, 'TRIS', {"pos": vertices}, indices=indices)


def draw():
    shader.uniform_float("color", (0, 0.5, 0.5, 1.0))
    batch.draw(shader)


bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_PIXEL')
```

2D Image

To use this example you have to provide an image that should be displayed.

```
import bpy
import gpu
from gpu_extras.batch import batch_for_shader

IMAGE_NAME = "Untitled"
image = bpy.data.images[IMAGE_NAME]
texture = gpu.texture.from_image(image)

shader = gpu.shader.from_builtin('IMAGE')
batch = batch_for_shader(
```

```
batch = batch_for_shader(
    shader, 'TRI_FAN',
    {
        "pos": ((100, 100), (200, 100), (200, 200), (100, 200)),
        "texCoord": ((0, 0), (1, 0), (1, 1), (0, 1)),
    },
)


def draw():
    shader.bind()
    shader.uniform_sampler("image", texture)
    batch.draw(shader)


bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_PIXEL')
```

```
"""
3D Image
--------

Similar to the 2D Image shader, but works with 3D positions for the image vertices.
To use this example you have to provide an image that should be displayed.
"""
import bpy
import gpu
from gpu_extras.batch import batch_for_shader

IMAGE_NAME = "Untitled"
image = bpy.data.images[IMAGE_NAME]
texture = gpu.texture.from_image(image)

shader = gpu.shader.from_builtin('IMAGE')
batch = batch_for_shader(
    shader, 'TRIS',
    {
        "pos": ((0, 0, 0), (0, 1, 1), (1, 1, 1), (1, 1, 1), (1, 0, 0), (0, 0, 0)),
        "texCoord": ((0, 0), (0, 1), (1, 1), (1, 1), (1, 0), (0, 0)),
    },
)


def draw():
    shader.uniform_sampler("image", texture)
    batch.draw(shader)


bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_VIEW')
```

Generate a texture using Offscreen Rendering

1. Create an `gpu.types.GPUOffScreen` object.
2. Draw some circles into it.
3. Make a new shader for drawing a planar texture in 3D.
4. Draw the generated texture using the new shader.

4. Draw the generated texture using the new shader.

```
import bpy
import gpu
from mathutils import Matrix
from gpu_extras.batch import batch_for_shader
from gpu_extras.presets import draw_circle_2d

# Create and fill offscreen
########################################

offscreen = gpu.types.GPUOffScreen(512, 512)

with offscreen.bind():
    fb = gpu.state.active_framebuffer_get()
    fb.clear(color=(0.0, 0.0, 0.0, 0.0))
    with gpu.matrix.push_pop():
        # reset matrices -> use normalized device coordinates [-1, 1]
        gpu.matrix.load_matrix(Matrix.Identity(4))
        gpu.matrix.load_projection_matrix(Matrix.Identity(4))

        amount = 10
        for i in range(-amount, amount + 1):
            x_pos = i / amount
            draw_circle_2d((x_pos, 0.0), (1, 1, 1, 1), 0.5, segments=200)


# Drawing the generated texture in 3D space
############################################

vert_out = gpu.types.GPUStageInterfaceInfo("my_interface")
vert_out.smooth('VEC2', "uvInterp")

shader_info = gpu.types.GPUShaderCreateInfo()
shader_info.push_constant('MAT4', "viewProjectionMatrix")
shader_info.push_constant('MAT4', "modelMatrix")
shader_info.sampler(0, 'FLOAT_2D', "image")
shader_info.vertex_in(0, 'VEC2', "position")
shader_info.vertex_in(1, 'VEC2', "uv")
shader_info.vertex_out(vert_out)
shader_info.fragment_out(0, 'VEC4', "FragColor")

shader_info.vertex_source(
    "void main()"
    "{"
    "  uvInterp = uv;"
    "  gl_Position = viewProjectionMatrix * modelMatrix * vec4(position, 0.0, 1.0);"
    "}"
)

shader_info.fragment_source(
    "void main()"
    "{"
    "  FragColor = texture(image, uvInterp);"
    "}"
)
```

```
)

shader = gpu.shader.create_from_info(shader_info)
del vert_out
del shader_info

batch = batch_for_shader(
    shader, 'TRI_FAN',
    {
        "position": ((-1, -1), (1, -1), (1, 1), (-1, 1)),
        "uv": ((0, 0), (1, 0), (1, 1), (0, 1)),
    },
)


def draw():
    shader.uniform_float("modelMatrix", Matrix.Translation((1, 2, 3)) @ Matrix.Scale(3, 4)
    shader.uniform_float("viewProjectionMatrix", bpy.context.region_data.perspective_matri
    shader.uniform_sampler("image", offscreen.texture_color)
    batch.draw(shader)


bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_VIEW')
```

Copy Off-screen Rendering result back to RAM

This will create a new image with the given name. If it already exists, it will override the existing one.

Currently almost all of the execution time is spent in the last line. In the future this will hopefully be solved by implementing the Python buffer protocol for `gpu.types.Buffer` and `bpy.types.Image.pixels` (aka `bpy_prop_array`).

```
import bpy
import gpu
import random
from mathutils import Matrix
from gpu_extras.presets import draw_circle_2d

IMAGE_NAME = "Generated Image"
WIDTH = 512
HEIGHT = 512
RING_AMOUNT = 10


offscreen = gpu.types.GPUOffScreen(WIDTH, HEIGHT)

with offscreen.bind():
    fb = gpu.state.active_framebuffer_get()
    fb.clear(color=(0.0, 0.0, 0.0, 0.0))
    with gpu.matrix.push_pop():
        # reset matrices -> use normalized device coordinates [-1, 1]
        gpu.matrix.load_matrix(Matrix.Identity(4))
        gpu.matrix.load_projection_matrix(Matrix.Identity(4))

        for i in range(RING_AMOUNT):
            draw_circle_2d(
                (random.uniform(-1, 1), random.uniform(-1, 1))
```

```
                (random.uniform(-1, 1), random.uniform(-1, 1)),
                (1, 1, 1, 1), random.uniform(0.1, 1),
                segments=20,
            )

    buffer = fb.read_color(0, 0, WIDTH, HEIGHT, 4, 0, 'UBYTE')

offscreen.free()


if IMAGE_NAME not in bpy.data.images:
    bpy.data.images.new(IMAGE_NAME, WIDTH, HEIGHT)
image = bpy.data.images[IMAGE_NAME]
image.scale(WIDTH, HEIGHT)

buffer.dimensions = WIDTH * HEIGHT * 4
image.pixels = [v / 255 for v in buffer]
```

Rendering the 3D View into a Texture

The scene has to have a camera for this example to work. You could also make this independent of a specific camera, but Blender does not expose goo functions to create view and projection matrices yet.

```
import bpy
import gpu
from gpu_extras.presets import draw_texture_2d

WIDTH = 512
HEIGHT = 256

offscreen = gpu.types.GPUOffScreen(WIDTH, HEIGHT)


def draw():
    context = bpy.context
    scene = context.scene

    view_matrix = scene.camera.matrix_world.inverted()

    projection_matrix = scene.camera.calc_matrix_camera(
        context.evaluated_depsgraph_get(), x=WIDTH, y=HEIGHT)

    offscreen.draw_view3d(
        scene,
        context.view_layer,
        context.space_data,
        context.region,
        view_matrix,
        projection_matrix,
        do_color_management=True)

    gpu.state.depth_mask_set(False)
    draw_texture_2d(offscreen.texture_color, (10, 10), WIDTH, HEIGHT)
```

```
bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_PIXEL')
```

Custom Shader for dotted 3D Line

In this example the arc length (distance to the first point on the line) is calculated in every vertex. Between the vertex and fragment shader that value is automatically interpolated for all points that will be visible on the screen. In the fragment shader the `sin` of the arc length is calculated. Based on the result a decision is made on whether the fragment should be drawn or not.

```
import bpy
import gpu
from random import random
from mathutils import Vector
from gpu_extras.batch import batch_for_shader

vert_out = gpu.types.GPUStageInterfaceInfo("my_interface")
vert_out.smooth('FLOAT', "v_ArcLength")

shader_info = gpu.types.GPUShaderCreateInfo()
shader_info.push_constant('MAT4', "u_ViewProjectionMatrix")
shader_info.push_constant('FLOAT', "u_Scale")
shader_info.vertex_in(0, 'VEC3', "position")
shader_info.vertex_in(1, 'FLOAT', "arcLength")
shader_info.vertex_out(vert_out)
shader_info.fragment_out(0, 'VEC4', "FragColor")

shader_info.vertex_source(
    "void main()"
    "{"
    "  v_ArcLength = arcLength;"
    "  gl_Position = u_ViewProjectionMatrix * vec4(position, 1.0f);"
    "}"
)

shader_info.fragment_source(
    "void main()"
    "{"
    "  if (step(sin(v_ArcLength * u_Scale), 0.5) == 1) discard;"
    "  FragColor = vec4(1.0);"
    "}"
)

shader = gpu.shader.create_from_info(shader_info)
del vert_out
del shader_info

coords = [Vector((random(), random(), random())) * 5 for _ in range(5)]

arc_lengths = [0]
for a, b in zip(coords[:-1], coords[1:]):
    arc_lengths.append(arc_lengths[-1] + (a - b).length)

batch = batch_for_shader(
    shader, 'LINE_STRIP',
    {"position": coords, "arcLength": arc_lengths},
)
```

```
def draw():
    matrix = bpy.context.region_data.perspective_matrix
    shader.uniform_float("u_ViewProjectionMatrix", matrix)
    shader.uniform_float("u_Scale", 10)
    batch.draw(shader)


bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_VIEW')
```

Custom compute shader (using image store) and vertex/fragment shader

This is an example of how to use a custom compute shader to write to a texture and then use that texture in a vertex/fragment shader. The expected resu is a 2x2 plane (size of the default cube), which changes color from a green-black gradient to a green-red gradient, based on current time.

```
import bpy
import gpu
from mathutils import Matrix
from gpu_extras.batch import batch_for_shader
import time

start_time = time.time()

size = 128
texture = gpu.types.GPUTexture((size, size), format='RGBA32F')

# Create the compute shader to write to the texture
compute_shader_info = gpu.types.GPUShaderCreateInfo()
compute_shader_info.image(0, 'RGBA32F', "FLOAT_2D", "img_output", qualifiers={"WRITE"})
compute_shader_info.compute_source('''
void main()
{
  vec4 pixel = vec4(
    sin(time / 1.0),
    gl_GlobalInvocationID.y/128.0,
    0.0,
    1.0
  );
  imageStore(img_output, ivec2(gl_GlobalInvocationID.xy), pixel);
}''')
compute_shader_info.push_constant('FLOAT', "time")
compute_shader_info.local_group_size(1, 1)
compute_shader = gpu.shader.create_from_info(compute_shader_info)

# Create the shader to draw the texture
vert_out = gpu.types.GPUStageInterfaceInfo("my_interface")
vert_out.smooth('VEC2', "uvInterp")
shader_info = gpu.types.GPUShaderCreateInfo()
shader_info.push_constant('MAT4', "viewProjectionMatrix")
shader_info.push_constant('MAT4', "modelMatrix")
shader_info.sampler(0, 'FLOAT_2D', "img_input")
shader_info.vertex_in(0, 'VEC2', "position")
shader_info.vertex_in(1, 'VEC2', "uv")
shader_info.vertex_out(vert_out)
```

```
shader_info.fragment_out(0, 'VEC4', "FragColor")

shader_info.vertex_source(
    "void main()"
    "{"
    "  uvInterp = uv;"
    "  gl_Position = viewProjectionMatrix * modelMatrix * vec4(position, 0.0, 1.0);"
    "}"
)

shader_info.fragment_source(
    "void main()"
    "{"
    "  FragColor = texture(img_input, uvInterp);"
    "}"
)

shader = gpu.shader.create_from_info(shader_info)

batch = batch_for_shader(
    shader, 'TRI_FAN',
    {
        "position": ((-1, -1), (1, -1), (1, 1), (-1, 1)),
        "uv": ((0, 0), (1, 0), (1, 1), (0, 1)),
    },
)


def draw():
    shader.uniform_float("modelMatrix", Matrix.Translation((0, 0, 0)) @ Matrix.Scale(1, 4)
    shader.uniform_float("viewProjectionMatrix", bpy.context.region_data.perspective_matri
    shader.uniform_sampler("img_input", texture)
    batch.draw(shader)
    compute_shader.image('img_output', texture)
    compute_shader.uniform_float("time", time.time() - start_time)
    gpu.compute.dispatch(compute_shader, 128, 128, 1)


def drawTimer():
    for area in bpy.context.screen.areas:
        if area.type == 'VIEW_3D':
            area.tag_redraw()
    return 1.0 / 60.0


bpy.app.timers.register(drawTimer)
bpy.types.SpaceView3D.draw_handler_add(draw, (), 'WINDOW', 'POST_VIEW')
```