Improving p5py

A GSOC 2020 proposal

Introduction

Motivation

p5py is a native python library that provides a python interface to the Processing language. While Python Mode does exist in the Processing Development Environment, it was built with Jython, which has seen some difficulties in supporting python 3. With python 2 being already deprecated by the Python Foundation, now is a great time to improve p5py as a potential replacement for Python Mode.

Building a native python library has several advantages, including but not limited to:

1. Easily interface with other existing python libraries (e.g. numpy, scipy, pytorch)
2. Succinct, expressive, and beginner-friendly syntax

However, many features are in the Processing Language but are missing in p5py. Furthermore, the p5py API is slightly different from that of Processing. This proposal (and consequently the GSOC project) aims to address those problems.

Base Goals

1. Standardize p5py API so it's as close as possible to that of the Processing Language (Java Mode) while being reasonably pythonic (e.g. supporting some python-only features and syntax)
2. Explore moving tessellation of shapes to OpenGL for improved performance
3. Improve 3D support

Stretch Goals

1. Investigate the performance and development-difficulty tradeoff between OpenGL and Vulkan.
2. Add support for live coding of sketches through the python REPL.
3. Investigate porting popular Processing libraries to p5py. For instance, adding video and audio support.
4. Write a preprocessor that converts Processing (Java) code to p5py, giving the user warning on not-yet-implemented APIs and automatically converting camelCase to snake_case.
5. Improve automated testing
6. Adapt `bezier_vertex`, `quadratic_vertex`, `curve_vertex` for 3D.

Design

Below, I will talk about the details as well as the design choices I made for each Base Goal.

Standardize the p5py API

Most of the p5py's design-level API changes from the Processing Language can be found here, and I agree with the majority of them. Nevertheless, below are a few items that I think could be changed to improve usability. *If you want to discuss them and/or propose additional changes, feel free to leave a comment!*

- `PShape` function signatures

"With the exception of the `point()` functions, all drawing functions that allow the user to pass in coordinates using tuples." -- p5py documentation

If I were to implement a new drawing language from scratch, I would have preferred to have every shape function signature to only accept vectors or vector-like objects. However, given that p5py is the Python counterpart of the Processing language, I think the benefits of interoperability and consistency outweigh the benefits of syntactic

cleanness. Furthermore, the current design is not particularly consistent in that some arguments are expecting vector-like objects while other arguments are not, leading to potential confusion. For example:

1. As mentioned in the documentation, `point` does not use vector-like arguments
2. In `'CORNER'` mode, `rect` expects the `location` to be a vector-like object while `width` and `height` to be scalars, leading to functions calls like `rect((0, 0), 100, 45)`

Ideally, we could support both types of calls if python had function signature overloading. Given that this isn't the case, we could still achieve similar results by checking the type of the argument being passed in (A discussion of the topic can be found at #130). If argument-checking turns out to be too slow, I propose that we should at least refactor the function signatures for shapes to match the API of the Processing Language.

The benefits of this approach include that new users will be able to copy-and-paste code from the original Processing Language, change camelCase to snake_case, and run their code. Furthermore, existing users of the original Processing Language will face fewer difficulties when they switch to p5py. Last but not least, using scalars everywhere is also consistent with the Processing Language's Python Mode.

- `push_matrix` and `pop_matrix`

Currently, `push_matrix` is implemented as a context-manager that automatically pops the transformation matrix at the end of a `with` block. While this is very pythonic and introduces great readability in most cases, this could lead to code that's too far indented to the right when multiple `with` blocks are nested. Furthermore, this introduces additional work when the user attempts to adapt their code from the Processing Language to p5py. Therefore, I propose retaining `push_matrix`'s ability of being a context manager while making it usable as a normal function, much like python's built-in `open` function. In addition, I will implement a `pop_matrix` function corresponding to the `popMatrix` function in Processing.

Concretely, users will have the option to choose from

```
with push_matrix():
    do_transformation()
```

and

```
push_matrix()
do_transformation()
pop_matrix()
```

Implementation Note: Although matrices are stored naturally in the context manager's "stack" in the current implementation, we may have to introduce a global stack data structure that stores transformation matrices to support the latter push-pop syntax.

Explore moving tessellation to OpenGL

Current triangulation implementation in p5py

The existing implementation uses the `triangle` python module to triangulate shapes before sending them to the GPU. The `triangle` module is in turn a wrapper around the Triangle C library written by Jonathan Shewchuk that implements the Delaunay triangulation algorithm.

This site notes that the Triangle Library by Prof. Shewchuck has several deficiencies. In particular,

1. "It does not like duplicate vertices or duplicate edges. 'Duplicate' in this case is relative to numeric precision: For a building 10-100 meters in size, two vertices within 8cm of each other, defining a very short edge, can cause Triangle to crash."

2. “It does not like it when a hole (inner ring) in the polygon has a vertex in the same location as one in the outer ring (crash).” --- vterrain.org

So let’s use OpenGL?

From first glance, using OpenGL is as simple as using the following API:

```
gluTessBeginPolygon(tess, user_data);
    gluTessBeginContour(tess);
        gluTessVertex(tess, coords, vertex_data);
        ...
        gluTessVertex(tess, coords_n, vertex_data);
    gluTessEndContour(tess);
gluTessEndPolygon(tess);
```

(Code example adapted from songho.ca)

However, further searches indicated that this API is not part of OpenGL core, but part of GLU (OpenGL Utility Library). According to GLFW

“GLU has been deprecated and should not be used in new code.”

Doing a quick search in the source of python glfw bindings, I couldn’t find the tessellation APIs listed. Therefore, it isn’t immediately clear as to how to use the tessellation API from GLFW.

Another option is to use the newer tessellation API in OpenGL 4.0, which is significantly more verbose than the old GLU API because it can also perform subdivision operations. Note that this API is also not present in the python glfw bindings.

Triangulation in p5.js and Processing

Maybe we can reference the triangulation implementations in other branches of the processing language? Here’s what I found:

For Processing (OpenGL backend), GLU was referenced through the JOGL binding, evidenced by source files PGraphicsOpenGL.java#L10985 and PJOGL.java#L605. Given

that we don't have JOGL on Python and GLFW doesn't support GLU, we might have to look for other options.

For p5.js, tessellation support was achieved via libtess.js, which uses the GLU tessellation algorithm but rewritten in JavaScript (reference).

PyOpenGL

Seeing how prevalent the GLU tessellation algorithm is, I started looking for Python libraries that implemented it. Turns out, I didn't find an implementation but a wrapper inside PyOpenGL.

Since we already have `vispy.gloo` for making OpenGL calls, I'm a little hesitant to introduce another library that has more or less the same functionality. It's good to know that this is another option though.

Vispy

I asked the question of how to use GLU bindings on the vispy Gitter chat after not finding anything related to GLU on their documentation. I will update this document when I receive a response.

Update:

> "Short answer is no. Long answer: I'm not familiar with the gluTess API specifically, but I'm guessing this has to do with tessellation shaders, right? Currently VisPy does not have any support for tessellation shaders but I think there are PRs trying to add it. In the last year we added support for geometry shaders which really set up the interfaces for adding additional shaders like the tessellation shaders, but so far these have not been added." —@djhoese

Scipy

Scipy also has a function that performs the Delaunay triangulation algorithm. Instead of wrapping around Prof. Shewchuck's library, it wraps around Qhull

Conclusion

After looking at all the above options, it isn't clear to me that using GLU/OpenGL is 100% the superior option compared to the current implementation. Therefore, I plan to start with this portion of the project by profiling the existing code to see how slow (or fast) `triangle` is. If it is a bottleneck, I will attempt to replace `triangle` with other drop-in options like `scipy` and repeat the profiling step. I could also modify the code to take care of simple tessellations (e.g. tessellating a rect) by hand and only call a library when we need to do something more advanced (e.g. filling bezier curves) if profiling results show that the library wrappers present a significant overhead. If all those fail, or if we need PyOpenGL for other functionality, I will then give GLU a try.

Update: we may have to switch out `triangle` if we want to properly support `beginShape()` and `endShape()` in 3D even if profiling proves that performance is not an issue. See the discussion below.

Improve 3D Support

Currently, the `0.6.0` update brought several 3D functions to `p5py`. As of now, rotation, translation, and normal 2D drawing calls seem to work in `P3D` mode. However, when 3D primitives are involved, an error is thrown (see #141).

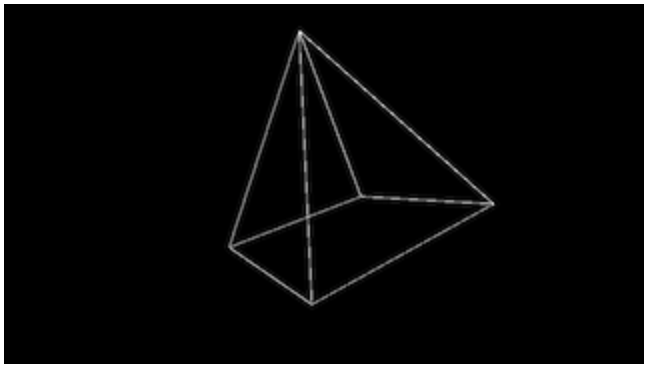
Update: while looking at the code, I found a quick fix that makes 3D primitives work (see #149).

3D Tessellation

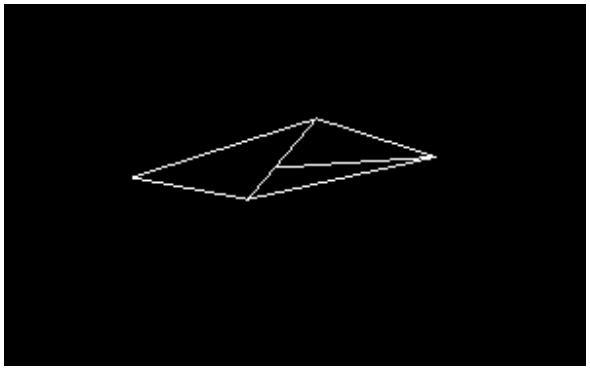
As of now `beginShape()` and `endShape()` do not seem to work properly in p5py's `P3D`. Take the example script and convert it to python:

```
from p5 import *

def setup():
    no_loop()

def draw():
    size(640, 360);
    background(0);

    translate(width/2, height/2, 0);
    stroke(255);
    rotate_x(PI/2);
    rotate_z(-PI/6);
    no_fill();

    begin_shape();
    vertex(-100, -100, -100);
    vertex( 100, -100, -100);
    vertex(   0,    0,  100);

    vertex( 100, -100, -100);
    vertex( 100,  100, -100);
    vertex(   0,    0,  100);

    vertex( 100, 100, -100);
    vertex(-100, 100, -100);
    vertex(   0,   0,  100);

    vertex(-100,  100, -100);
    vertex(-100, -100, -100);
    vertex(   0,    0,  100);
    end_shape();

run(mode='P3D')
```

We expect the output to be



but instead get



From looking at the code, I noticed that all the vertices are being passed to the 2D class `PShape` instead of the 3D class `Geometry`, regardless of the renderer that's in use. (Also, there seems to be an effort to ignore `vertex` calls in 3D mode but was not implemented correctly). Since `PShape` uses a 2D triangulation library, it doesn't know how to handle 3D points properly, resulting in the square we see in the above image.

To solve this problem, I propose two options:

1. If we only want to support geometries "without holes" (i.e. `POINTS`, `LINES`,`LINE_STRIP`,`LINE_LOOP`,`TRIANGLES`,`TRIANGLE_STRIP`, `TRIANGLE_FAN`) in 3D, we can hand-code each case and store the correct primitives in a `Geometry` object.
2. If we want to support geometries "with holes" (e.g. a cone with its top cut off), we may have to look for another triangulation/tessellation library that supports 3D.

Deciding which option to take and whether there are better options available probably requires some community discussion.

Implement `normal_material()`

The is a function that exists in p5.js but not in Processing. It assigns a color to a pixel solely based on the normal vector of the fragment being rendered. Useful for debugging, so this will be one of the first shaders that I'll implement.

Implement `basic_material(r, g, b)`

This is the default material when `fill` is called. Returns a uniform color.

An aside on lighting and materials API

The p5.js material API is not particularly flexible in that it forces the user to choose between individual materials like `ambientMaterial` and `specularMaterial` when in reality all of the materials can be united under the Blinn-Phong model.

The Processing Language offers better flexibility in offering individual `ambient`, `emissive`, `shininess`, and `specular` calls. However, I think `emissive` is rather a misnomer because the material itself does not act as a light but reflects light from light sources. Given that `emissive` modifies the color of diffuse reflections in the Blinn-Phong model, I propose renaming it to `diffuse`. Aside from this change, everything else carries over from the Processing Language material API. Below is the API that I am proposing.

Implement `ambient(r, g, b)`

Sets the ambient light color reflected by the material. This is sometimes called the ambient coefficient.

Implement `diffuse(r, g, b)`

Sets the diffuse light color reflected by the material.

Implement `shininess(p)`

Sets the amount of gloss of the material. It is the exponential above the cosine term in Blinn-Phong

Implement `specular(r, g, b)`

Sets the specular light color reflected by the material.

Implement `blinn_phong_material(r, g, b)`

This is the material being applied whenever `ambient`, `diffuse`, `shininess`, or `specular` is called.

Blinn-Phong shading can be decomposed into three parts: ambient, diffuse, and specular.

The ambient component is essentially a constant term that is always present. We calculate it by summing all the ambient lights in a scene and multiplying it with the normalized ambient coefficient set by `ambient`.

The diffuse component takes the normal vector of a surface into account and varies how much light is reflected depending on the angle that the surface makes with the incoming light.

The specular component not only accounts for the direction of the light (like the diffuse component) but also the direction of the viewer. If the viewer is not on the path of the reflected light, the specular component falls off quickly, producing the glossy reflections we see on some materials.

The color shown on the screen by the GPU is the sum of all three components. Here's a nice visualization of the different components.

Implement `lights()`

This is a wrapper for setting up default lights

```
def lights():
    ambient_light(128, 128, 128)
    directional_light(128, 128, 128, 0, 0, -1)
    light_falloff(1, 0, 0)
```

Implement `ambient_light(r, g, b)`

Adds an ambient light to the list of lights. Participates in ambient lighting calculations.

Implement `directional_light(r, g, b, x, y, z)`

Adds a directional light to the list of lights. Participates in diffuse & specular lighting calculations.

Implement `point_light(r, g, b, x, y, z)`

Adds a point light to the list of lights. Participates in diffuse & specular lighting calculations.

Implement `light_falloff(constant, linear, quadratic)`

Sets the falloff rates for point lights and ambient lights that have locations.

d = distance from light position to vertex position

falloff = 1 / (CONSTANT + d * LINEAR + (d*d) * QUADRATIC)

Note that like the Processing Language, directional lights are not affected because directional lights don't have a location associated with them, only a direction. This is in turn because we only get parallel light rays that are like directional lights in nature when the light source is very far away (e.g. the sun).

Does not implement `light_specular(r, g, b)`

The specular component should be defined by the color of the light (which is defined when creating the light) and the material it hits. Therefore, I found it unnecessary to include another function to override the specular color of a light specifically. This being said, if there is a demand, this function can still be implemented without too much work.

Timeline

- Community Bounding Period
  Seek feedback and potentially more requests for API standardization
  Seek feedback for triangulation/tessellation libraries to use
  Seek feedback for whether to support drawing shapes with holes in 3D
  Seek feedback for the slightly revamped material and lighting API
- Week 1-2
  Refactor PShape function signatures
  Implement `pop_matrix` and make `push_matrix` individually callable.
  Implement other API standardization tasks if requested by the community in the Community Bounding Period
  Update documentation and tests for the affected APIs.
- Week 3-4
  Write sample scenes for profiling and gather initial profiling results
  Replace `triangle` with `scipy` and measure performance improvements
  Replace `triangle` with `GLUTess` and measure performance improvements
  If this changes the API, update the documentation.
  Update tests.
- Week 5-6
  Make a decision on whether to support drawing shapes with holes in 3D, which would impact which triangulation library that we will use.
  Finish integrating the triangulation library of choice.
  Adapt `begin_shape`, `end_shape`, and `vertex` for 3D
  Stretch goal: also adapt `bezier_vertex`, `quadratic_vertex`, `curve_vertex` for 3D.
- Week 7-8
  Implement `normal_material`
  Implement `basic_material`
  Start implementing `blinn_phong_material` and related methods `ambient`, `diffuse`, `shininess`, `specular`.
  Start implementing light related methods: `lights`, `ambient_light`, `point_light`, `directional_light`, `light_falloff`.

- Week 9-10
  Continue with the work in week 7-8
  Add documentation and tests for the newly added APIs
- Week 10-12
  Continue improving documentation and testing.
  Build the final version and publish on pypi.
  Flexible time reserved for schedule overruns.

About me

I am an undergraduate student studying computer science at UC Berkeley and a software developer at Lawrence Berkeley Lab. At the lab, I work with the DESI team on image processing and performance optimization. Most recently, we are working on porting our Python code to CUDA so that it can run on the GPU.

I like computer graphics and maintain a blog that contains some of the previous projects that I've done, including a path tracer, a CPU rasterizer, and a mesh editor. I first encountered processing last semester when I took an intro to art class and was very drawn to the idea of enabling everyone to express their creativity through code. Therefore, I decided to apply to the processing foundation as soon as I saw it being listed on GSOC 2020.

I have been part of the open-source community for many years, having both written my own projects and submitted occasional patches to others. If you are interested, feel free to check out my Github. In the process of writing this proposal, I happened to find a few simple fixes for either the documentation and the code and submitted PRs #147, #148, #149. At the time of writing they have not been merged, so please take a look and maybe provide some feedback :) I look forward to having a fun and productive summer working with you all!