Abstract Algebra, Tessellations, and Shaders

Using GLSL and Abstract Algebra to Render Simple Tesselations of the Plane

Devin Pohl

Colorado State University

MATH 366: Introduction to Abstract Algebra

Dr. Hortensia Soto, Dr. Jessi Lajos

Sunday, Dec 5, 2021

Abstract

This work relates the problem of procedurally rendering tessellations with concepts from abstract algebra. After narrowing the domain of the problem and picking a specific example, a naive implementation with several issues is presented. Then, concepts from abstract algebra are applied in order to simplify and improve the process. Finally, an optimized implementation is presented. Throughout this paper, GLSL shader code is used to generate images in order to set implementation goals and restrictions, as well as show progress. This work concludes with a brief discussion on generalizing the presented solution, as well as how it could potentially be optimized further.

Contents

List of Listings

List of Figures

Introduction

Objective

Tessellations are beautiful, complex patterns that arise from a simple set of rules. However, they are hard to realize procedurally. This means that for applications such as shaders, programs that generate color and texture in computer graphics, efficient tessellations are out of reach for a naive programmer. Luckily, using concepts from abstract algebra, significant improvements and optimizations can be made on tessellation shaders. Presented in this text is the approach of a naive programmer, followed by optimizations from abstract algebra, and finally a demonstration of an ideal implementation. Throughout this work, examples will be given of shader code, along with its outputs to help visualize the process of improvement. While this work does not present a *perfect* solution, it improves a particular part of the implementation process which is widely applicable to many classes of tessellations and shaders. But first, the definition and narrowing of scope for both shaders and tessellations are in order.

What are Shaders?

As described by Iché (2021), shaders are small programs that run directly on a graphics card (GPU). They are optimized to run extremely quickly, running hundreds or even thousands of computations in parallel on different elements. While elements in a shader are typically pixels, they can also be vertices of a 3D object, outputs of other shaders, or computations entirely unrelated to graphics. For the scope of this paper, only pixel shaders will be considered, as they are the most simple to understand and straightforward to demonstrate. Note however that topics described in this paper apply to other types of shaders as well.

Shaders are implemented via programming. This can be done with raw GPU assembly, but is more commonly done with frameworks, more easily understood abstractions that compile down to assembly. Popular frameworks include GLSL, HLSL, and GPGPU. For this paper, GLSL will be used – its similarity with the widely recognized C programming language allows it to be generally understood by a wide audience. An example of GLSL shader code can be found in Listing 1, adapted from Feng (2021). When rendered, this pixel shader produces Figure 1.

```
vec3 mandel(vec2 z0) {
  float k = 0.0;
  vec2 z = vec2(0.0);
  for(int i = 0; i < 420; ++i) {
    z = vec2(z.x*z.x-z.y*z.y, z.x*z.y*2.0) + z0;
    if (length(z) > 20.0) break;
    k += 1.0;
  }
  float mu = k + 1.0 - log2(log(length(z)));
  return sin(mu*0.1 + vec3(0.0,0.5,1.0));
}
void main() {
  float ar = iResolution.x / iResolution.y;
  vec2 uv = gl_FragCoord.xy / iResolution.yy
    - vec2(0.66 * ar, 0.5);
  uv = uv * 2.0 + vec2(-0.3, 0.0);
  float p = 10.0;
  float t = mod(13.0, p);
  if (t > p/2.0) t = p - t;
  float scale = 0.5 + pow(2.0, t);
  vec2 offset = vec2(-1.36799, .01);
  uv += offset*scale;
  uv /= scale;
  fragColor = vec4(mandel(uv), 1.0);
}
```

Listing 1
GLSL sample code

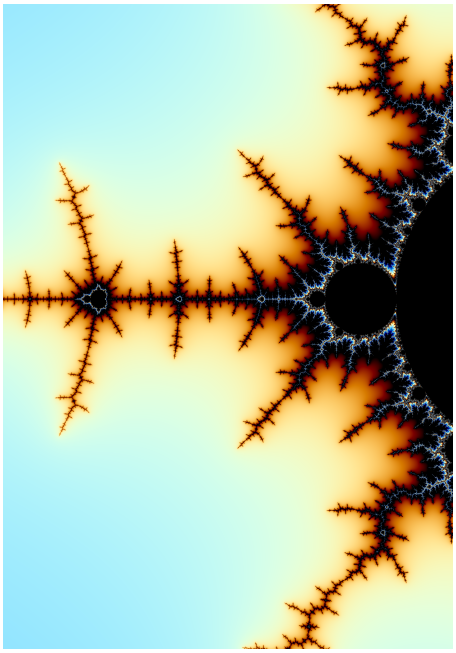


Figure 1
Rendering of the sample code

What are Tessellations?

Tessellations, or tilings, are described by Grunbaum and Shephard (1990) as methods of filling a space without gaps via patterned placement of geometric shapes. One such example can be found in Figure 2, a tiling described by Penrose (1974). For the sake of simplicity, the only tessellations directly considered in this paper are those based on the $p1$ wallpaper group – Fedorov (1891) defines these as tilings which only use a single shape under simple translation. Figure 3 presents an example, a tessellation rendered by Milkins (2011) by sliding around a single shape. This class of tiling will be referred to as *simple tessellations* throughout this paper. Such tight restrictions will allow for clear and efficient application of abstract algebra. The end of this paper will briefly show how the approach presented in this paper applies to several other classes of tessellations as well.

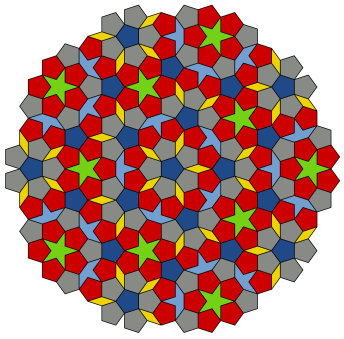


Figure 2
A Penrose Tiling

Figure 3
A wallpaper group

A Naive Implementation

In order to fully appreciate this application of abstract algebra, it helps to see an implementation free from optimization. This section will present a standard method of tessellation that an untrained programmer is likely to use. The steps of deriving its algorithm will be given in the order that a programmer is likely to take; this will highlight the short-sightedness of an implementation that does not use abstract algebra. Once this implementation is presented, its various flaws will be examined and the search for an optimized implementation will begin.

Step 1: Defining the Base Shape

The first step to generating a tessellation is producing the base shape. This can be done in several ways, but considering that the shape will shortly render in GLSL, the method used should answer the fundamental question: *is some arbitrary point within the base shape or not?* This question is so important due to GLSL's nature of computing each pixel entirely independently of all other pixels. With this in mind, let us pick a sample shape to tessellate.

Choosing a Shape

The shape chosen for a sample is important. In many ways, it acts as a test of the implementation software. As such, the shape should be chosen to highlight and extinguish programming bugs before they occur. One potential bug can be found in a lazy modulus-based calculation, where the programmer assumes that for any arbitrary point, the center of its associated tile is closer to any other center. This can be highlighted by having a shape with both convex and concave portions. Another potential bug is treating both the x and y axes as the same. This can be highlighted by choosing a tessellation which tiles in a skewed fashion. With these constraints in mind, Figure 4 presents a good sample shape.

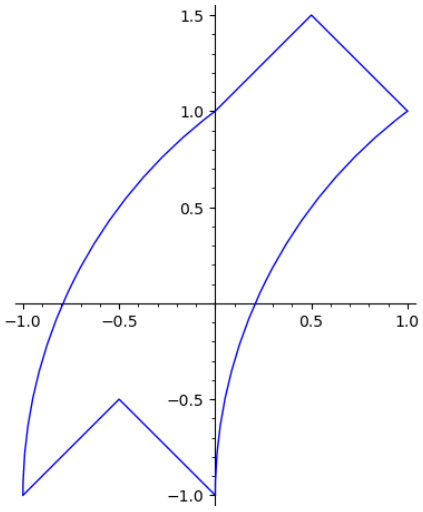


Figure 4
Sketch of the sample tile

Realizing in Set Notation

Following the previous question of *is some arbitrary point within the base shape or not*, it is exceedingly useful to represent the sample shape in set notation. This is because set notation directly expresses the answer to this question. For example, the set $\{(x, y) : (x, y) \in \mathbb{R}^2, x > 0\}$ describes the right half of the plane, and the answer to our question is then: *is* $x > 0$? Unfortunately, our sample shape is not so simple; it is a complex area defined by complex interactions. In a way, this is good for the integrity of this test case – a complex shape will give rise to challenges in implementation that may be encountered in the field.

With that said, there are three parts of the shape to consider: the band between arcs, the union of the upper triangle, and the removal of the bottom triangle. The first component, the arc, can be represented as all points inside the circle formed by the left-most arc and outside of the circle formed by the right-most arc with the requisite y coordinate: $A = \{(x, y) : (x, y) \in \mathbb{R}^2, y > -1 \wedge y < 1 \wedge \sqrt{(x - 1.5)^2 + (y + 1)^2} < 2.5 \wedge \sqrt{(x - 2.5)^2 + (y + 1)^2} > 2.5\}$. The next component, the upper triangle, is a bit more simple to express with regions formed around lines: $B = \{(x, y) : (x, y) \in \mathbb{R}^2, y > 1 \wedge x + 1 > y \wedge -x + 2 > y\}$. Finally, the lower triangle can be realized in a similar way: $C = \{(x, y) : (x, y) \in \mathbb{R}^2, y > -1 \wedge x > y \wedge -x - 1 > y\}$. Interactions between these three regions produce the base tile: $(A - C) \cup B$.

Rendering the Base Tile

With the shape definition out of the way, what remains is implementing its definition in GLSL. Luckily, GLSL pixels are based on Cartesian coordinates: pixel coordinates are integers, with x starting at 0 on the left side of the output buffer and y starting at 0 on the top of the output buffer. Little setup is needed to convert this scheme to the plane from Figure 4. The code in Listing 2 provides some boilerplate code to convert the given coordinates `gl_FragCoord` into $\mathbb{R}^2$ coordinates stored in the global variable `cc`. This code will be implicitly included in all further listings in this paper.

```
uniform float x_width = 3.0;
vec2 center_of_buffer =
  vec2(iResolution.x/2,
       iResolution.y/2);
float scale = iResolution.x/x_width;

vec2 get_cartesian_coord() {
  vec2 c = gl_FragCoord.xy
    - center_of_buffer.xy;
  c.y *= -1.0; // GLSL places the y
  ↪  axis backwards, so flip it
  return c/scale;
}

vec2 cc = get_cartesian_coord();
```

Listing 2
Boilerplate code to provide Cartesian coordinates

```
bool is_in_band(vec2 coord) {
  // A = {(x,y) : (x,y) ∈ ℝ², y > −1 ∧ y < 1 ∧ √((x − 1.5)² + (y + 1)²) <
  ↪  2.5 ∧ √((x − 2.5)² + (y + 1)²) > 2.5}
  bool in_left = sqrt(pow(coord.x-1.5,2)+pow(coord.y+1,2)) < 2.5;
  bool outside_right = sqrt(pow(coord.x-2.5,2)+pow(coord.y+1,2))
    > 2.5;
  return (coord.y > -1.0) && (coord.y < 1.0)
    && in_left && outside_right;
}
bool is_in_upper_triangle(vec2 coord) {
  // B = {(x,y) : (x,y) ∈ ℝ², y > 1 ∧ x + 1 > y ∧ −x + 2 > y}
  return (coord.y > 1.0) && (coord.x+1.0>coord.y)
    && (-coord.x+2>coord.y);
}
bool is_in_lower_triangle(vec2 coord) {
  // C = {(x,y) : (x,y) ∈ ℝ², y > −1 ∧ x > y ∧ −x − 1 > y}
  return (coord.y > -1.0) && (coord.x>coord.y)
    && (-coord.x-1>coord.y);
}
bool is_in_base_tile(vec2 coord) {
  bool A = is_in_band(coord);
  bool B = is_in_upper_triangle(coord);
  bool C = is_in_lower_triangle(coord);
  return (A && !C) || B; // (A − C) ∪ B
}
void main() {
  fragColor = is_in_base_tile(cc) ? vec4(0.0, 0.0, 0.0, 1.0)
    : vec4(1.0, 1.0, 1.0, 1.0);
}
```

Listing 3
GLSL shader for the sample tile

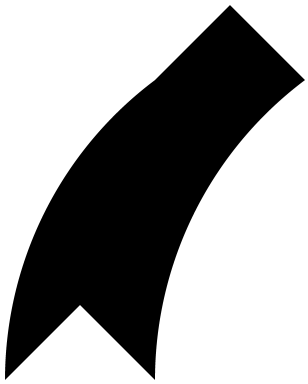


Figure 5
Rendering of the sample tile

What remains is determining if any given (normalized Cartesian) coordinate falls within the base tile. Following the set notation given in the previous subsection, implementation is fairly straightforward. Simply check whether the current coordinate `cc.xy` is in any of the regions A, B, or C, and apply boolean logic to determine inclusion in the base tile. Listing 3 provides the code to do so, and the output is rendered in Figure 5. As can be seen, the base tile is rendered properly. As an additional note, the function `is_in_base_tile`, which combines all the above checks, will also be reused in future listings for brevity.

Step 2: Tiling the Base Shape

Following the previous work, the next step is to determine whether or not an arbitrary point is in an arbitrary tile. GLSL does not make this task easy; because branches or loops – the most basic methods of making decisions – are not supported, the only way forward is through modulus arithmetic. Listing 4 presents what is essentially the only solution to this problem under

these constraints. The basic structure of this code is to shift the coordinate grid so that the entire base tile has entirely non-negative coordinates, compute the modulus of the current coordinate to place it into the bounding box of the base tile, shift back the coordinate system, and then check if the remaining coordinate is in the base tile. This approach has a few issues, which will be discussed later.

```
uniform vec2 tile_max_dimensions =
  vec2(1.0, 2.5);
uniform vec2 tile_lower_left_corner =
  vec2(-1.0, -1.0);
bool in_offset(vec2 coord, vec2
↪  offset, vec2 repeat) {
  coord -= tile_lower_left_corner;
  coord -= offset;
  coord = mod(coord,
  ↪  tile_max_dimensions * repeat);
  coord += tile_lower_left_corner;
  return is_in_base_tile(coord);
}
```

Listing 4
Determining if any point is in a given tile

The next step is getting the center coordinate of a tile. As this will be discussed in great detail during the optimization section of this paper, a brief explanation of this concept will suffice. The base tile is associated with the coordinate $(0, 0)$. The tile immediately to the right of the base tile is associated with the coordinate $(1, 0)$, and so on. Getting the tile coordinate is important because it allows for distinguishing between tiles, and coloring them differently. Listing 5 gives what is essentially the only solution to this problem with the given tools.

With all of the groundwork laid, it is finally time to present a proper tessellation shader. Listing 6 presents the code to do this. Its structure is fairly simple: establish a few x-coordinate bands and assign different colors to them. This shortcut works because the top of the $(0, 0)$ tile does not touch the bottom of the $(0, 1)$ tile – a better approach will be discussed later in this paper. The output of this shader can be seen in Figure 6.

```
bool in_x(vec2 coord, float x_offset,
↪  int repeat) {
  bool r0 = in_offset(coord,
  ↪  vec2(x_offset, 0.0),
  ↪  vec2(repeat, 3));
  bool r1 = in_offset(coord,
  ↪  vec2(x_offset, 2.0),
  ↪  vec2(repeat, 3));
  bool r2 = in_offset(coord,
  ↪  vec2(x_offset, -2.0),
  ↪  vec2(repeat, 3));
  return r0 || r1 || r2;
}
```

Listing 5
Determining the x coordinate of a tile's center

Problems With This Approach

The naive implementation presented above has several problems, which culminate into one fatal design flaw. This flaw will be explained in this section, and used as a target to improve during optimization. The issue becomes apparent in Listing 6 with the existence of the variables `x0`, `x1`, and `x2`. These three variables track whether or not the current coordinate has a tile coordinate of 0, 1, or 2 (all modulus 3). The fact that these are all separate boolean variables –

```
void main() {
  bool x0 = in_x(cc, 0.0, 3);
  bool x1 = in_x(cc, 1.0, 3);
  bool x2 = in_x(cc, 2.0, 3);

  fragColor = x0 ? vec4(0.969, 0.569, 0.863, 1.0) :
    x1 ? vec4(0.486, 0.976, 0.537, 1.0) :
    x2 ? vec4(0.976, 0.929, 0.506, 1.0) :
    vec4(1.0, 1.0, 1.0, 1.0);
}
```

Listing 6
GLSL shader for naive tessellation

Figure 6
Rendering of the naive implementation

and not one integer tracking the tile coordinate – is the heart of the design flaw. With this implementation, it is **impossible to get the tile coordinate in linear complexity**. Because it is only possible to ask whether or not a given coordinate is in a given tile (thanks to Listing 4), the absolute tile coordinate cannot be asked for – only the *modulus* of the tile coordinate. The base of this modulus is also an issue – the higher the base the more computational complexity. For example, with a modulus base of five (which would allow five unique colors instead of the three in Figure 6), variables `x0` through `x4` would be required, and additional computation for each would be required. If the y tile coordinate is also required to properly color the tessellation, than the complexity increase would become quadratic. Clearly, an optimized solution needs to put the goal of finding the tile coordinate first.

An Optimized Implementation

In order to have a better implementation than the naive one given above, the tile coordinate needs to be computed with linear complexity. This, fascinatingly, *cannot be done without abstract algebra.* In this section, this statement will be proven, and its implications will be used to construct a new algorithm free from the errors above. Finally, this algorithm will be implemented in GLSL to produce an optimized shader.

Defining Tile Coordinates

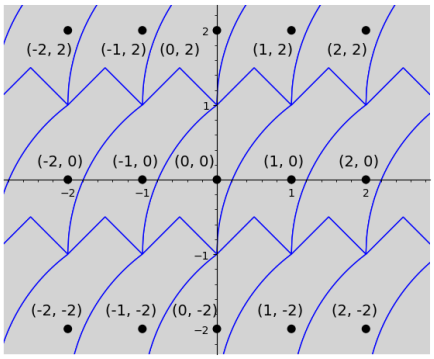


Figure 7
A visualization of tile coordinates

In order to better find tile coordinates, it helps to define them more clearly. For the purposes of this paper, the tile coordinate is the coordinate by which the base tile has been translated by to yield the current tile. Figure 7 illustrates this concept. Abstract algebra gives us the tools to define this concept more generally. Let T be a set of points that form a tile, and let S be the set of all tile coordinates. For example, in Figure 7, $S = \{(x, y) : x \in \{...-1, 0, 1, ...\}, y \in \{...-2, 0, 2, ...\}\}$. To tile the plane, repeatedly make copies of T, shifting it by every element in S. For the case of Figure 7, one such example is $T_{(1,2)} = \{(x_T + 1, y_T + 2) : (x_T, y_T) \in T\}$. The set of all tiles $T_S = \{T_s : s \in S\}$ completely fills the plane – this is the fundamental requirement of a simple tessellation. That is to say, $\bigcup T_S = \mathbb{R}^2$. In abstract algebra, this is called a *partition*. The fact that simple tessellations form a partition of the plane will be exceedingly useful in creating optimized shader code.

Characterizing the Naive Implementation

With tile coordinates properly explained, the naive implementation can be revisited. The goal of this section is to abstract out all the implementation details and explain *why* the naive implementation works (and why it fails). After sufficient abstraction, the exact problem with the naive implementation will become abundantly clear.

Asking for a Tile Coordinate

One of the fundamental operations of the naive implementation is asking *does coordinate s fall in tile T_s?* This operation is shown in Listing 4. Let this operation be g. Its problem, explained in more detail above, is that g cannot actually be implemented in GLSL for $\mathbb{R}^2$, only a small subset of $\mathbb{R}^2$. That is to say, $g(c)$ cannot directly exist in GLSL, only $g(m(c))$, where m is an additional modulus step. The workaround operation m is realized in the naive implementation according to the `repeat` parameter of `in_x` in Listing 5 – which essentially ignores all inputs for which g does not give a good answer.

Modulating the Plane

The other fundamental operation of the naive implementation is taking the modulus of a coordinate. This is done in line 8 of Listing 4, when the coordinate modulus is taken. Let this operation be f. As an important note, f is *required* in the naive implementation due to the lack of loops in GLSL.

Putting the Two Together

Given the above definitions of f, g, and m, the basic structure of the shader is $f(g(m(c)))$, where c is the current coordinate. In order to have a better implementation, g – or a function very similar to it – *must* be the final function or else information is lost. In the following section, concepts from abstract algebra will be used to turn $f(g(m(c)))$ into $g(f(c))$, allowing the programmer to *indirectly* ask what tile any particular coordinate is a part of.

Endomorphisms

In abstract algebra, a homomorphism is a mapping between two groups. An endomorphism is a homomorphism that maps from a group to itself. Endomorphisms of Abelian groups have a peculiar property, noted by Pinter (2010), where for any two endomorphisms h and j on an Abelian group, $h \circ j = j \circ h$.

It remains to be shown that the above definitions of f and g are endomorphisms. As for g (a version *without* flaws), Pinter (2010) shows that for a group G and a partition G/H, G/H is a homomorphic image of G. And in the case of simple tessellations, the set of all tiles is isomorphic to the set of all tile coordinates. Because tile coordinates live in $\mathbb{R}^2$, this means that $\mathbb{R}^2$ is endomorphic to the set of all tile coordinates. Thus g is an endomorphism on $\mathbb{R}^2$. The proof for f is similar; because the modulus operation on $\mathbb{R}^2$ produces a partition of rectangles over $\mathbb{R}^2$, then f is an endomorphism on $\mathbb{R}^2$.

Finally, the critical optimization can take place. Because f and g are endomorphisms on the Abelian group $\mathbb{R}^2$, the naive approach can be improved from $f(g(m(c)))$ to $g(f(m(c)))$. Furthermore, because f and m are both modulation operations, m is redundant in this case and the shader implementation can be characterized simply as $g(f(c))$. And because g is acting on a subset of $\mathbb{R}^2$, it can efficiently be implemented in GLSL. The only caveat left is that $g(f(c))$ can

only give the coordinate of the tile containing particular point *relative to that point*. This is a non issue, as getting the non-modulated center coordinate of any particular tile can simply be done as $c + g(f(c))$. With the critical issue of the naive implementation removed, a final implementation can begin.

Visualizing the New Implementation

Before writing any GLSL code, it is useful to visualize the above optimization. Figure 8 presents a visualization of $g(f(c))$, where every point in the rectangle is associated with a vector. This vector, when added back to the original c, yields the absolute tile coordinate. It is derived directly from the $g(f(c))$ algorithm: modulate coordinates, then associate with a tile.

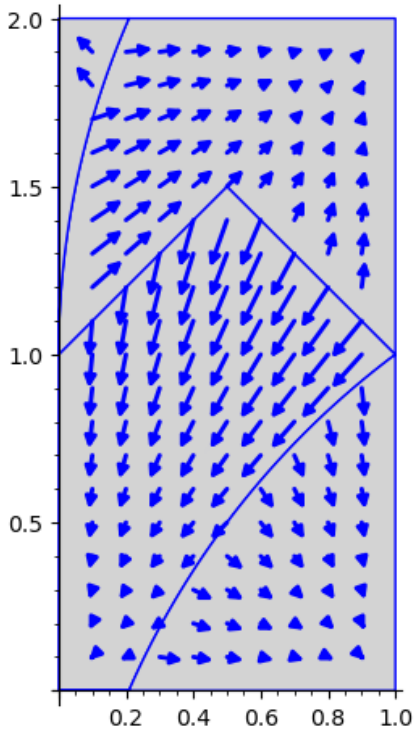


Figure 8
Visulization of vector mapping

However, the modulation step requires a bit of additional explanation. Both the x and y coordinates are modulated to form the rectangle in Figure 8. The base of each modulus operation determines the width and height of this rectangle. If these dimensions are too high, then more work is required to query the relative tile coordinate, as there will be more regions in the rectangle. If the dimensions are too small, then aliasing occurs and the tessellation falls apart. As it turns out, the optimal dimensions for this rectangle can be found by cutting up and rearranging a tile. Notice that rearranging the individual regions of Figure 8 yields exactly one copy of the base tile. A rigorous proof of this is outside the scope of this paper, as is a discussion of *where* to place this rectangle to improve performance further. As such, Figure 8 will be directly used as a guideline for implementation.

Final Implementation

With the mathematics behind simple tessellation thoroughly explored, it is now time to write GLSL code again. The first step, f, is modulating the plane into the

```
1  uniform vec2 rect = vec2(1.0, 2.0);
2  vec2 mod_cc = vec2(mod(cc, rect));
```

Listing 7
Modulating the plane

rectangle formed in Figure 8. This is done quite simply in Listing 7. The next step is to use the geometry present in the base tile to determine a modulated point's relative tile coordinate. This is done in Listing 8. While the implementation of this step seems rather complicated, its details can largely be ignored for purposes of this paper. This is due to the fact that all the equations present are determined solely by the geometry of the base tile and can be derived ahead of time or even automatically by another program that generates GLSL code from tile geometry. Listing 8 also finishes the main task of calculating the absolute tile coordinate according to the relative tile coordinate.

```
bool inside_triangle =
  (mod_cc.x+1.0>mod_cc.y)
  && (-mod_cc.x+2>mod_cc.y);
bool inside_upper_curve =
  sqrt(pow(mod_cc.x-2.5,2)
       + pow(mod_cc.y-1,2)) < 2.5;
bool inside_lower_curve =
  sqrt(pow(mod_cc.x-2.5,2)
       + pow(mod_cc.y+1,2)) < 2.5;

vec2 tile_cc_relative =
  inside_lower_curve ? vec2(1, 0)
  : inside_triangle ? vec2(0, 0)
  : inside_upper_curve ? vec2(1, 2)
  : vec2(0, 2);

vec2 tile_coord() {
  vec2 tile_cc_relative =
    inside_lower_curve ? vec2(1, 0)
    : inside_triangle ? vec2(0, 0)
    : inside_upper_curve ? vec2(1, 2)
    : vec2(0, 2);
  return cc - mod_cc + tile_cc_relative;
}
```

Listing 8
Segmenting the rectangle

What remains is applying the newly optimized algorithm. While interesting output can be generated in many ways, Listing 9 randomly colors each tile using the tile coordinate as the seed. As can be seen in Figure 9, this produces desirable results. For reference, Appendix A has the expanded form of Listing 9 without any parts omitted. Appendix B has a larger output of the shader to showcase its functionality. This concludes development on the optimized solution.

```
// The canonical GLSL random function
float rand(vec2 co){
    return fract(sin(dot(co, vec2(12.9898, 78.233))) *
    ↪  43758.5453);
}

void main( void ) {
  vec2 abs_tile_coord = tile_coord();

  float R  = rand(abs_tile_coord);
  float G  = rand(abs_tile_coord+3.14);
  float B = rand(abs_tile_coord+2.72);

  fragColor = vec4(R, G, B, 1.0);
}
```

Listing 9
Tiling the plane with pseudorandom colors

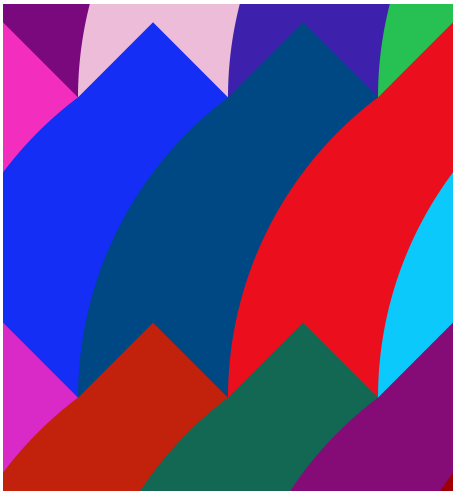


Figure 9
Rendering of the final implementation

Conclusion

Further Optimizations

While the optimized implementation above solves several of the problems present in the naive implementation, it is not perfect. There are other potential improvements whose implementations are outside the scope of this paper, but worth noting as topics for future study. The first is that rendered images are prone to aliasing, a phenomenon where at low resolutions the borders of objects look choppy. This can be seen in Figure 10, which presents the output of Listing 9 except rendered at a native resolution of 40 by 30 pixels. Anti-aliasing is the study and mitigation of this phenomenon, which can be solved in the case of shaders with a *fragment* shader, which uses vector output instead of pixel output. Another potential improvement on this implementation is the automatic generation of the segmentation equations from Listing 8. An industry-standard approach to this is to use a higher level programming language which can natively manipulate vector objects, most often a Lisp, to generate GLSL-compatible equations which are then compiled along-side the rest of the optimization. This process of converting from a higher level programming language to a lower level one is called *compiling*, and often includes further performance optimizations that a programmer is unlikely to notice or employ. These two further optimizations are just the tip of the iceberg; shader efficiency has been studied for decades and will only continue to become more rigorous.

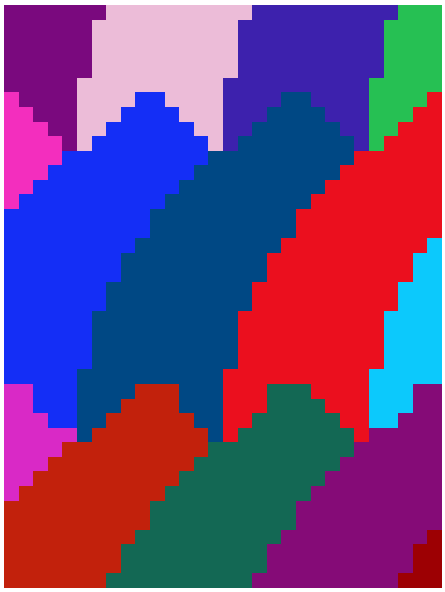


Figure 10
Aliasing in action

Other Types of Tessellations

As briefly mentioned in the introduction of this text, the type of tessellations studied in this paper are from the $p1$ wallpaper group. This type of tiling uses a single shape and a single orientation to yield a tiling. Other wallpaper groups are relatively straightforward to optimize using similar methods. In each of these cases, the method to doing so is just making the rectangle

from Figure 8 more complicated. For example, the $p2$ group allows rotations of 180° and can be optimized by declaring the two base tiles as one meta-tile. Then, this meta-tile can be cut up and rearranged into a rectangle. So long as the tile coordinates are properly kept track of for each base tile in the meta-tile, the implementation remains the same as in this paper. Indeed, this approach works for all wallpaper groups. In fact, it actually works for all periodic tessellations, thanks to the properties discussed in the endomorphisms section of this work. With only minor modifications, the approach discussed in this paper can be used to generate even more complex tilings.

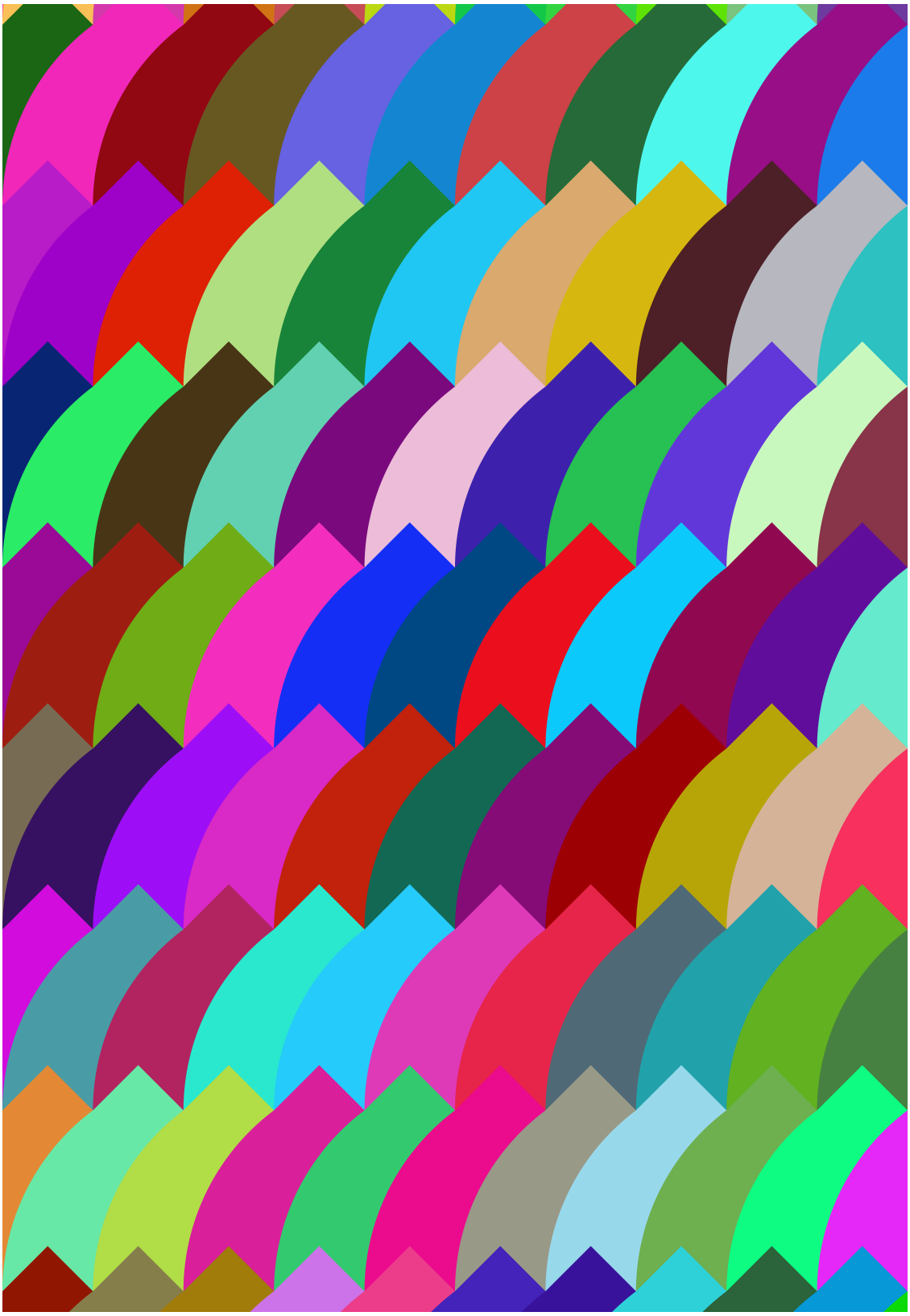
Summary

Procedurally rendering tessellations is a difficult problem in computer graphics. This problem was tackled in this paper, by first presenting a naive implementation, and then using concepts of abstract algebra to produce a greatly improved implementation. In the end, elegant shader code yielded highly desirable results. While the topics in this paper were focused on the example at hand, it was briefly shown how these topics widely apply to a number of adjacent problems, as well as how they could be used as a foundation for further optimization. Overall, the problem of rendering tessellations optimally is one where concepts of abstract algebra clearly play an important role.

Appendix A: Full Shader Code

```
uniform float x_width = 10.0;
vec2 center_of_buffer =
  vec2(iResolution.x/2,
       iResolution.y/2);
float scale = iResolution.x/x_width;

vec2 get_cartesian_coord() {
  vec2 c = gl_FragCoord.xy
    - center_of_buffer.xy;
  c.y *= -1.0; // GLSL places the y axis backwards, so flip it
  return c/scale;
}

vec2 cc = get_cartesian_coord();
uniform vec2 rect = vec2(1.0, 2.0);
vec2 mod_cc = vec2(mod(cc, rect));
bool inside_triangle =
  (mod_cc.x+1.0>mod_cc.y)
  && (-mod_cc.x+2>mod_cc.y);
bool inside_upper_curve =
  sqrt(pow(mod_cc.x-2.5,2)
       + pow(mod_cc.y-1,2)) < 2.5;
bool inside_lower_curve =
  sqrt(pow(mod_cc.x-2.5,2)
       + pow(mod_cc.y+1,2)) < 2.5;

vec2 tile_cc_relative =
  inside_lower_curve ? vec2(1, 0)
  : inside_triangle ? vec2(0, 0)
  : inside_upper_curve ? vec2(1, 2)
  : vec2(0, 2);

vec2 tile_coord() {
  vec2 tile_cc_relative =
    inside_lower_curve ? vec2(1, 0)
    : inside_triangle ? vec2(0, 0)
    : inside_upper_curve ? vec2(1, 2)
    : vec2(0, 2);
  return cc - mod_cc + tile_cc_relative;
}
// The canonical GLSL random function
float rand(vec2 co){
    return fract(sin(dot(co, vec2(12.9898, 78.233))) * 43758.5453);
}

void main( void ) {
  vec2 abs_tile_coord = tile_coord();

  float R  = rand(abs_tile_coord);
  float G  = rand(abs_tile_coord+3.14);
  float B = rand(abs_tile_coord+2.72);

  fragColor = vec4(R, G, B, 1.0);
}
```

Appendix B: Larger Output



References

Fedorov, E. (1891). Symmetry in the plane. 2, pages 345–390. Proceedings of the Imperial St. Petersburg Mineralogical Society.

Feng, L. (2021). ob-glsl github repository. `https://github.com/finalpatch/ob-glsl`. Online; accessed 30 November 2021.

Grunbaum, B. and Shephard, G. C. (1990). *Tilings and Patterns*. W.H. FREEMAN.

Iché, T. (2021). Shaders longform. *VFXDoc*. `https://vfxdoc.readthedocs.io/en/latest/`. Online; accessed 30 November 2021.

Milkins, J. (2011). Can someone explain the math behind tessellation? *Mathematics Stack Exchange*. `https://math.stackexchange.com/q/36834`. Online; accessed 30 November 2021.

Penrose, R. (1974). The role of aesthetics in pure and applied mathematical research. *Bulletin of the Institute of Mathematics and Its Applications*, 10(2):266–271.

Pinter, C. (2010). *A book of abstract algebra*. Dover Publications, Mineola, New York.