

UCSD CSE 167 Assignment 1: 2D Graphics

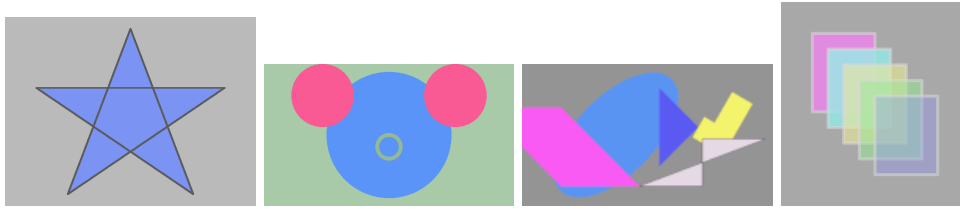


Figure 1: Images we will produce in this homework.

Computer graphics is the field that studies how to process *visual data*, such as shapes, volumes, and lights. **Images** are an important class of visual data: they can be the pictures recorded by the camera of your cellphone or a DSLR, outputs of video games or movie visual effects, legends and arrows on Google map telling you where to go next, or visualization of the radio waves coming from a black hole. An important subfield of computer graphics, rendering, studies the generation of images.

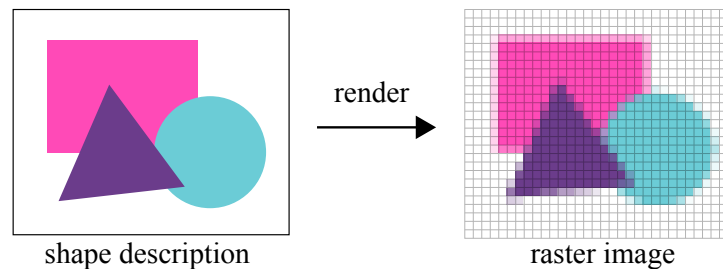


Figure 2: In this homework, we will render a set of 2D shapes into a raster image.

In our first homework, we will implement a 2D renderer that takes “shapes” (e.g., circle, lines, squares, and triangles), and turn them into a raster image (Fig. 2). A raster image is a particular kind of image that represent 2D contents using a grid of *pixels*, where each pixel denotes the color at that location. Raster images are convenient because our displays (and our camera sensor) usually also represent images as a grid of pixels.

Before you start coding, we recommend you to go through the whole handout to have some ideas of what needs to be done. Some of the instructions are **intentionally vague** because we want you to figure things out. Please come to our office hours or ask on Piazza if you have any questions.

Submission. Submit your code and the outputs of your code through Canvas. Please zip them without the temporary object files and binaries generated by the compiler.

Grading. We will compare your outputs to our reference solutions. You don’t have to be accurate at binary level (as there will be floating point cancellation errors), we allow some error margins (the TAs will check manually if something seems off).

Collaboration policy. We expect you to write the code on your own. Feel free to discuss between the peers and ask us questions though!

0 Building balboa

We are going to build our code on top of a currently barebone codebase *balboa*. Balboa already includes all the third party libraries (stb_image, stb_image_write, json, tinyply, GLFW, and glad) in its repository. All

you need to do is to clone the repo and build it using CMake (assuming you are in a Unix-like system):

```
git clone --recurse-submodules https://github.com/BachiLi/balboa_public
mkdir build
cd build
cmake ..
make -j
```

For Windows users, either directly loading `CMakeLists.txt` in your Visual Studio, or you can execute the commands above until `make -j`, and open the solution file. See Readme for more information.

Note that `balboa` uses submodule for the GLFW dependency. If you cloned the repository without the `--recurse-submodules` flag, you can download the submodule using

```
git submodule update --init --recursive
```

The commands below assume your current working directory is the `build` directory. After building, you should see an executable `balboa`. Try typing the following command:

```
./balboa -hw 1_1
```

It will generate an image `hw_1_1.png` that is completely white.

We recommend you to quickly read through `main.cpp` and `hw1.cpp` to understand the structure of the code.

1 Rendering a single circle (5 pts)

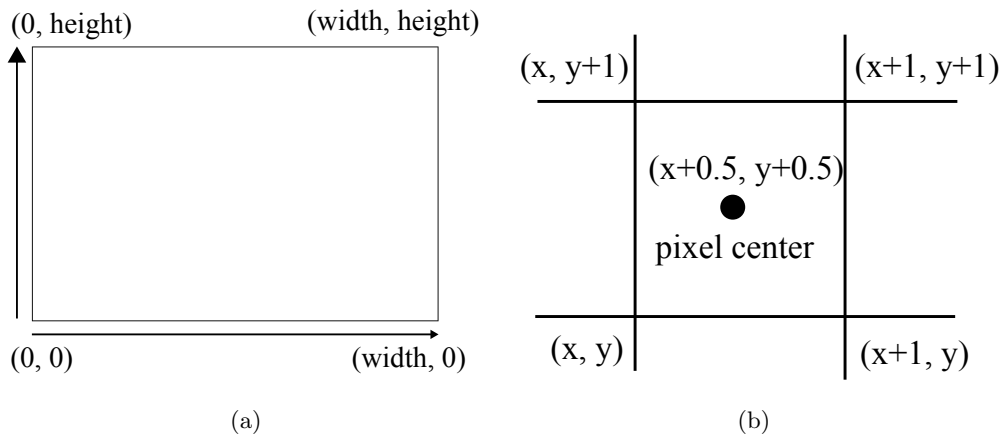


Figure 3: (a) The coordinate system of the “canvas” we will draw our shapes on. (b) The pixel center of for pixel (x, y) locates at $(x + 0.5, y + 0.5)$.

In the first part of this homework, we will render a single circle on our image. We are given a gray “canvas”, where in this part we assume it to be of size 640×480 . We need a **coordinate system** to talk about points on this canvas – Fig. 3a illustrates the coordinate system. We call this coordinate system the “canvas space”. We are further given the center, radius, and color of the circle. These are command line arguments that we will parse for you.

To render an image, we need to determine the color for each pixel. For this part, we focus on determining the color of the center of the pixel (Fig. 3b). To determine the pixel color, we decide whether the pixel center hits the circle or not. If it hits the circle, then we decide that the pixel’s color is the circle’s color. Otherwise, we decide that the pixel’s color is the background color (by default, let’s say it’s $(0.5, 0.5, 0.5)$). How to decide whether the pixel center hits the circle? That’s for you to figure out!

Balboa provides a few utilities that will be useful for this homework: `Vector2`, `Vector3`, and `Image3`. They are defined in `vector.h` and `image.h`.

Vectors. `Vector2` and `Vector3` are utilities for representing 2D and 3D vectors. You can access their members using `.x`, `.y`, and `.z`. We overloaded a few common operators so that you can add and subtract vectors, and multiply them with scalars:

```
Vector2 v0 = ..., v1 = ...;
v0 + v1; // vector addition
v0 - v1; // subtraction
v0 * Real(0.5); // multiply by a scalar
v0 / Real(0.5); // divide by a scalar
dot(v0, v1); // dot product between the two vectors
normalize(v0); // returns a vector with same direction as v0, but with magnitude of 1
length(v0); // return the magnitude of v0
Vector3 v2 = ..., v3 = ...;
cross(v2, v3); // cross product between v2 & v3
std::cout << v0 << std::endl; // print the vector
```

In balboa, we represent floating point numbers using the type `Real`. It is by default defined to be a `double`. It is designed such that when you want to switch to single precision float (maybe for less memory cost), you can switch a single line in `balboa.h`.

```
using Real = double;
```

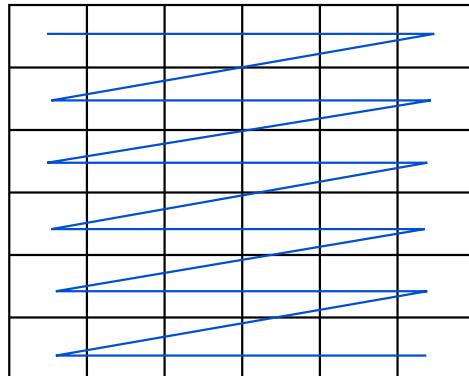


Figure 4: Storing a 2D image in an 1D array by scanning the image.

Images. `Image3` represents a 3-channel raster image (usually storing red, green, blue values in each channel respectively). We store the 2D image in a 1D array using a scanline from top left to bottom right (Fig. 4). **Note that the image coordinates might be different from the canvas coordinate system we specified earlier!** You can access the 2D image using the `()` operator:

```
Image3 img(640 /* width */, 480 /* height */);
Vector3 val = img(50, 20); // reading values from location (50, 20)
img(30, 60) = Vector3{0.5, 0.6, 0.7}; // writing values to location (30, 60)
img.width; // 640
img.height; // 480
imwrite("image.png", img); // write the image to the disk
Image3 img2 = imread3("image2.png"); // read an image from the disk
```

The pixel values are stored as `Real` in the memory. When we write the image into a file, the file format can often only supports 8-bit integers. The conversion is made in the `imwrite` function, where we apply a **gamma correction** to encode the image (with $\gamma = \frac{1}{2.2}$). We will likely talk about gamma correction in the class.

You should be ready to implement the circle rendering code in `hw_1_1` in `hw1.cpp` at this point. To see your results, in terminal, type:

```
./balboa -hw 1_1 -center 200 300 -radius 100 -color 0.3 0.7 0.5
./balboa -hw 1_1 -center -50 250 -radius 100 -color 0.7 0.3 0.3
./balboa -hw 1_1 -center 600 250 -radius 150 -color 0.3 0.5 0.7
./balboa -hw 1_1 -center 250 470 -radius 50 -color 0.7 0.5 0.7
```

or just any parameter you like! Fig. 5 shows our rendering for the first command.

In your submission, save the images generated by the second to fourth commands as

```
outputs/hw_1_1_1.png
outputs/hw_1_1_2.png
outputs/hw_1_1_3.png
```

We'll compare your output with our rendering for grading. Let's put the `outputs` folder in the main folder `balboa_public`.



Figure 5: Reference image for Homework 1.1.

If you see your circles, congratulations! If you have not done graphics-related stuff before, this is likely the first time you use code to generate an image from scratch. This is what makes computer graphics fun at least for me: you paint on a canvas using code (and sometimes math) instead of a brush. This makes people who are not that good in traditional art capable of generating beautiful pictures (yes, the images you generate will become prettier from now on). Furthermore, unlike things like Stable Diffusion or Veo, you maintain full control of the process: at this point, you know exactly how each pixel is generated, and if you don't like it, it's very easy to fix if you know how computer graphics works.

2 Rendering polylines (25 pts)

In this part, we will render a more complex shape called **polyline**, inspired by the Scalable Vector Graphics (SVG) standard. Therefore, what we describe below will be close to how some of the web browsers render these shapes. A polyline is a sequence of 2D points with straightlines connected through them. A *closed* polyline means that the last point is connected to the first point.

Similar to SVG, each polyline can have a *fill color* (only applies to closed polylines) and a *stroke color* (applies to both open and closed polylines) defined by an additional parameter stroke width. Fig. 6 illustrates

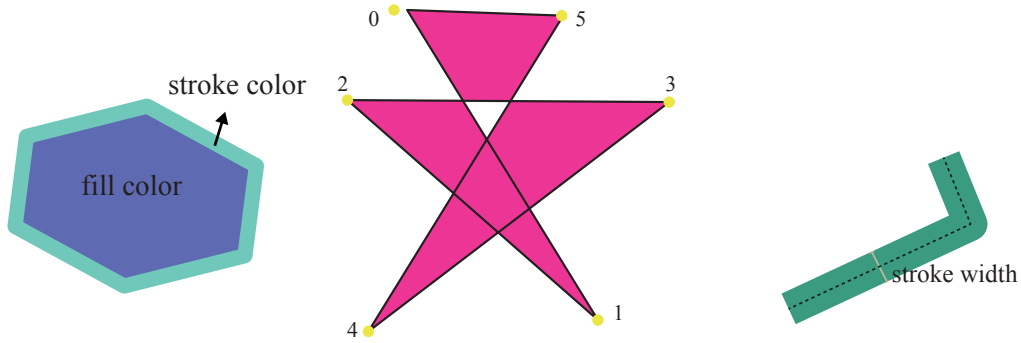


Figure 6: (a) Fill color vs. stroke color. (b) A more complex example for fill colors (the numbers next to the points are the order). (c) Stroke width for stroke color.

them (pay extra attention to a few details including the round corners of the star in Fig. 6(a), the middle regions in Fig. 6(b), and the flat caps of the line segment in Fig. 6(c)). We should have covered how to determine whether a point belongs to fill vs stroke regions in the lectures, but it could be fun to figure it out on your own too!

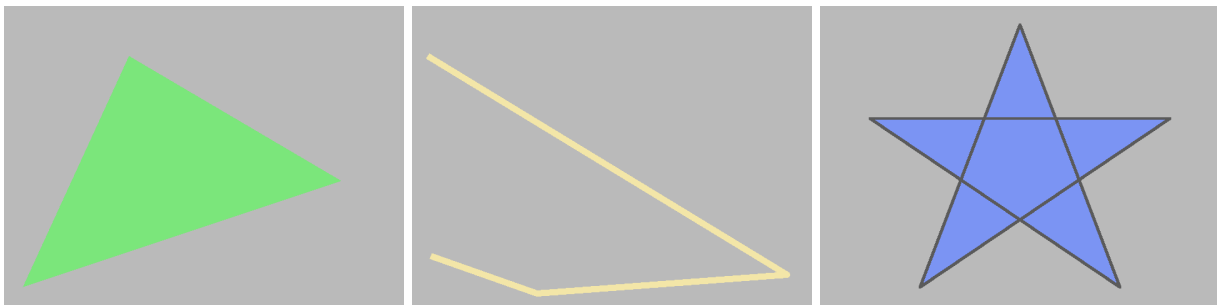


Figure 7: References for Homework 1.2.

To see your results, in terminal, type:

```
./balboa -hw 1_2 -points 30 30 540 200 200 400 --closed -fill_color 0.2 0.8 0.2
./balboa -hw 1_2 -points 25 400 600 50 200 20 30 80 -stroke_color 0.9 0.8 0.4 -stroke_width 10
./balboa -hw 1_2 -points 320 450 480 30 80 300 560 300 160 30 --closed -fill_color 0.2 0.3 0.9
-stroke_color 0.1 0.1 0.1 -stroke_width 5
```

Fig. 7 shows our rendering for the these three commands.

To test your code, we will use these commands:

```
./balboa -hw 1_2 -points 50 30 300 600 490 150 --closed -fill_color 0.8 0.2 0.8
./balboa -hw 1_2 -points 150 100 500 400 -stroke_color 0.2 0.2 0.9 -stroke_width 50
./balboa -hw 1_2 -points 50 30 300 600 490 150 --closed -stroke_color 0.8 0.2 0.8 -stroke_width 2
./balboa -hw 1_2 -points 50 400 500 30 480 420 20 40 --closed -fill_color 0.8 0.2 0.2
-stroke_color 0.2 0.2 0.2 -stroke_width 3
./balboa -hw 1_2 -points 250 420 300 400 200 350 450 300 150 250 500 200 100 150 550 100 320 50
320 5 -stroke_color 0.9 0.9 0.2 -stroke_width 20
./balboa -hw 1_2 -points 160 450 520 30 60 300 560 300 160 30 420 450 --closed -fill_color 0.9
0.4 0.2 -stroke_color 0.1 0.1 0.1 -stroke_width 4
```

In your submission, save the images generated by the commands above as

outputs/hw_1_2_1.png

outputs/hw_1_2_2.png
outputs/hw_1_2_3.png
outputs/hw_1_2_4.png
outputs/hw_1_2_5.png
outputs/hw_1_2_6.png

We will compare your output with our rendering for grading.

3 Rendering multiple shapes (15 pts)

Next, we will add the ability of rendering multiple shapes in our renderer. Before that, let's introduce our scene file format, so that we can easily create new scenes. There is no single standard way to represent a scene in computer graphics.¹ Thus, we will use an easily readable and parsable **JSON** format to describe our scene. It's easiest to explain by directly showing a scene:

```
{
  "resolution": [640,360],
  "background": [
    0.4, 0.6, 0.4
  ],
  "objects": [
    {
      "type": "circle",
      "center": [320,150],
      "radius": 30,
      "stroke_color": [0.3, 0.5, 0.3],
      "stroke_width": 10
    },
    {
      "type": "circle",
      "center": [490,280],
      "radius": 80,
      "fill_color": [0.95, 0.1, 0.3]
    },
    {
      "type": "circle",
      "center": [150,280],
      "radius": 80,
      "fill_color": [0.95, 0.1, 0.3]
    },
    {
      "type": "circle",
      "center": [320,180],
      "radius": 160,
      "fill_color": [0.1, 0.3, 0.95]
    }
  ]
}
```

The corresponding image is at the left of Fig. 8 (pay attention to the detail of the ordering of the shapes).

We will provide the scene parser for you. The scene is parsed into the following **struct**:

¹Though in 2D, **Scalable Vector Graphics (SVG)** is the most popular, and in 3D, studios are converging towards the **Universal Scene Description**.

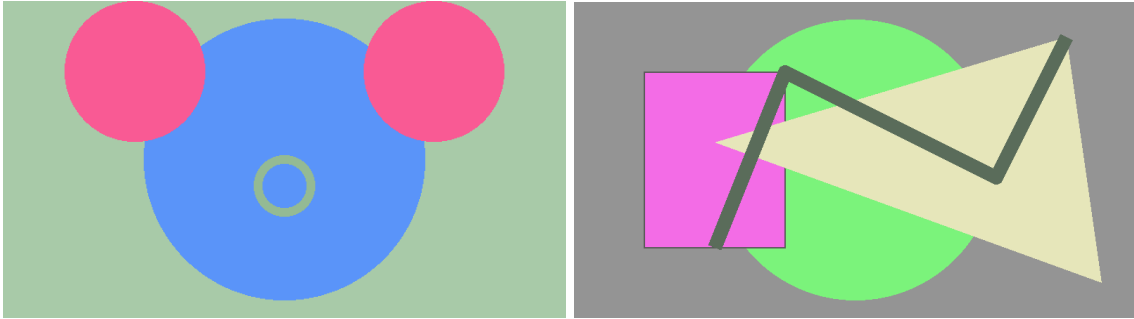


Figure 8: References for Homework 1.3.

```
struct Scene {
    Vector2i resolution;
    Vector3 background;
    std::vector<Shape> shapes;
};
```

where Shape is a `std::variant`:

```
struct Circle {
    Vector2 center;
    Real radius;
    std::optional<Vector3> fill_color;
    Real fill_alpha;
    std::optional<Vector3> stroke_color;
    Real stroke_alpha;
    Real stroke_width;
    Matrix3x3 transform;
};
```

```
struct Polyline {
    std::vector<Vector2> points;
    bool is_closed;
    std::optional<Vector3> fill_color;
    Real fill_alpha;
    std::optional<Vector3> stroke_color;
    Real stroke_alpha;
    Real stroke_width;
    Matrix3x3 transform;
};
```

```
using Shape = std::variant<Circle, Polyline>;
```

Don't worry about `transform` and `alpha` for now. We'll use it in the next part.

A `std::variant` can be seen as a *tagged union*:

```
struct Foo {
    int type;
    union {
        Type0 t0;
        Type1 t1;
        // ...
    }
};
```

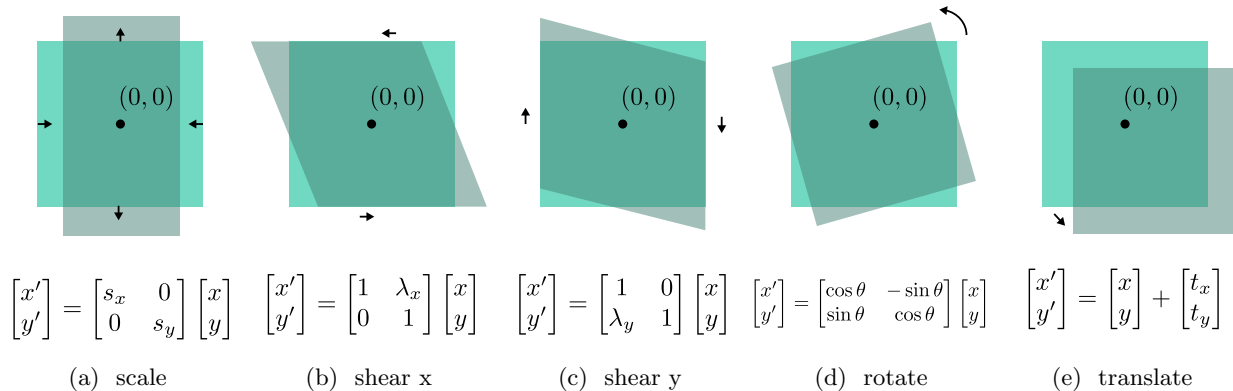


Figure 9: The five affine transformations we will support in this homework. Here we assume x axis is pointing right and y axis is pointing up. Notice that only the translation cannot be expressed using a 2×2 matrix.

Read more about it [here](#). I used variant here so that I can store all shapes in a single linear array without separate heap memory allocation.

In your code, one way to deal with different types of `Shape` could be the following:

```
if (auto *circle = std::get_if<Circle>(&shape)) {
    // do something with circle
} else if (auto *polyline = std::get_if<Polyline>(&shape)) {
    // do something with polyline
}
```

Our JSON scenes support (axis-aligned) rectangles and triangles, but they are converted to polylines by our scene parser, so handling circles and polylines should be sufficient.

Write your code in `hw_1_3` and test it using the following commands:

```
./balboa -hw 1_3 ../scenes/hw1/four_circles.json
./balboa -hw 1_3 ../scenes/hw1/four_shapes.json
```

We show our rendering of the two scenes in Fig. 8. We will grade your results using the following scenes:

```
./balboa -hw 1_3 ../scenes/hw1/five_shapes.json
./balboa -hw 1_3 ../scenes/hw1/cat.json
./balboa -hw 1_3 ../scenes/hw1/robot.json
```

In your submission, save your renderings of the scenes above as

```
outputs/hw_1_3_five_shapes.png
outputs/hw_1_3_cat.png
outputs/hw_1_3_robot.png
```

We will grade by comparing to our rendering.

4 2D transformation (15 pts)

Next, we're going to edit our shapes using some 2D transformations. These transformations can help us render more interesting shapes and generate animations. We are going to focus on *affine* transformations, where each point $x \in \mathbb{R}^2$ in the shape is transformed through the expression $Ax + b$ where A is some 2×2 matrix and b is a 2D vector.

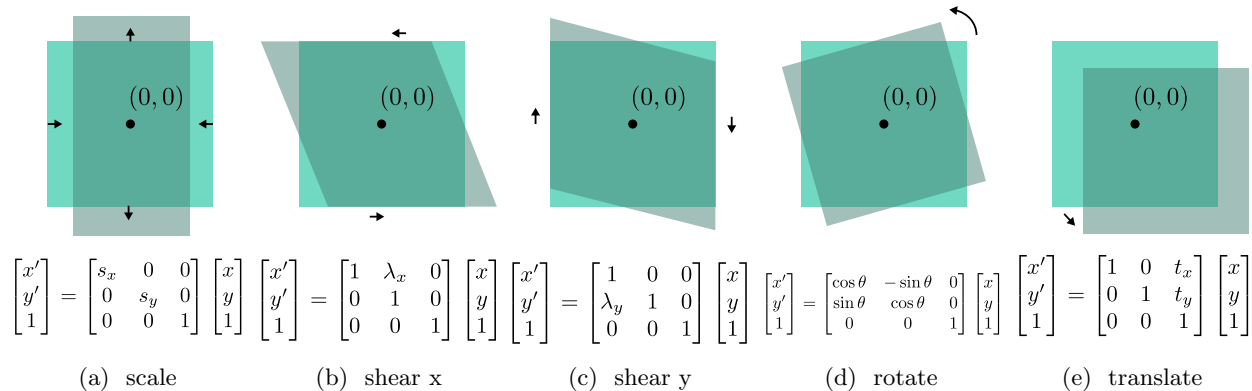


Figure 10: Using *augmented matrices*, we can represent all affine transforms, including translation, using 3×3 matrices. We recommend you to hand compute these matrix vector products and confirm they are the same as Fig. 9.

In this part, we will implement a few common affine transformations: scaling, shearing, rotation, and translation (in fact, all affine transformations can be written as combinations of these transformations, see “polar decomposition” if you are interested). Crucially, we also want to *combine* these transformations: for example, we might want to first rotate the shape, then translate it. As noted in Fig. 9, everything except for translation can be represented as a matrix vector product. This is annoying for combining the transformations: for all other transformations, we can combine them by multiplying the transformation matrices: for example, if we want to first rotate using matrix R then scale using matrix S , we can combine them into a single matrix SR (think: why isn’t it RS ?). The existence of translation makes this difficult. Fortunately, there is a very clever trick to unify every affine transformation into matrix vector products: observe that

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & b_0 \\ a_{10} & a_{11} & b_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (1)$$

is equivalent to

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \quad (2)$$

Therefore, we can represent all affine transformations using the 3×3 matrix above. Fig. 10 shows the corresponding version of the matrices for each transformation.

Another cool feature about augmented matrices is that they allow us to distinguish between *points* and *vectors*. For points, we set the third component to be 1. For vectors, we set the third component to be 0. This allows us to *turn off* translation for vectors: translating a vector should not affect its values!

To describe affine transformations in our scene files, for each object, we can have a `transform` property:

```
{
  "type": "circle",
  "fill_color": [0.3, 0.5, 0.8],
  "transform": [
    {"scale": [200,100]},
    {"rotate": 45},
    {"translate": [320,180]}
  ]
}
```

(Note that the rotation is described in degrees, not radians.) The circle has the default radius (1) and center $((0,0))$. The transformations are applied in the order they appear: the circle is first scaled, then rotated,

then translated (think: how would it look like?). Mathematically, if we denote the scaling matrix as S , the rotation matrix as R , and translation matrix as T , then the affine transformation matrix F is:

$$\text{translate}(\text{rotate}(\text{scale}(x))) = TRSx = Fx \quad (3)$$

The affine transformation matrix allows us to easily chain these operations and store the final result in a single matrix F .

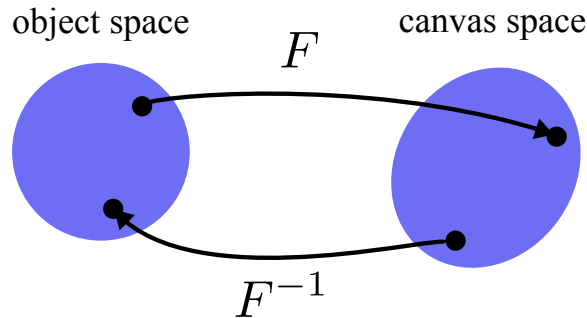


Figure 11: Transformation converts points between spaces.

The transformation matrix above can be seen as a relation between *spaces*. We call the space before the transformation the *object space* because this is where the space the original object is described in. Here, the transformation (let's call it F) converts points in the object space to the canvas space. On the other hand, the *inverse* of the transformation F^{-1} (i.e., matrix inversion) converts points in the canvas space to the object space. See Fig. 11 for a visualization.

The remaining question we need to address is: how do we test if the pixel center has hit the transformed shape? A main challenge is that the transformed shape may not be as easy to test compared to the original shape. For example, after linear transformations, a circle may become an ellipse. The trick we are going to test in the object space, instead of the canvas space. Given a canvas space point x representing a pixel's center, we convert it to the object space using the inverse transform F^{-1} , then test whether it hits the object after the inverse transform.

Matrices. `Matrix3x3` and `Matrix4x4` (defined in `matrix.h`) are balboa's matrix utilities. To access the i -th row and j -th column of a matrix m , you can use the `()` operator $m(i, j)$ (zero-based):

```
Matrix3x3 m; // initialize to all zeros
m(1, 2) = 1; // set row 1 column 2 to 1
Real m12 = m(1, 2); // get value from row 1 column 2
m = Matrix3x3::identity(); // set matrix to identity
m = inverse(m); // invert the matrix
Vector3 v = ...;
v = m * v; // matrix vector product
std::cout << m << std::endl; // print the matrix
```

Internally, balboa stores these matrices in the **column-major order**. This is to match the convention in OpenGL that we will use in Homework 3. This is realized in the implementation of the `operator()`:

```
T& operator()(int i, int j) {
    // Column major!
    return data[j][i];
}
```

To complete your implementation, first go to `hw1_scenes.cpp` and complete the function `parse_transformation`: this function parses the sequence of transformations and combines them into a 3×3 matrix. Next, go to `hw1.cpp` and finish your implementation to render transformed shapes.

Test your rendering using the following commands:

```
./balboa -hw 1_4 ../scenes/hw1/transformation.json
```

We show our rendering of the `transformation` scene in Fig. 12.

We will grade your results using the following scenes:

```
./balboa -hw 1_4 ../scenes/hw1/transformation_2.json
./balboa -hw 1_4 ../scenes/hw1/sun.json
./balboa -hw 1_4 ../scenes/hw1/piggy.json
```

In your submission, save your renderings of the scenes above as

```
outputs/hw_1_4_transformation_2.png
outputs/hw_1_4_sun.png
outputs/hw_1_4_piggy.png
```

We will grade by comparing to our rendering.

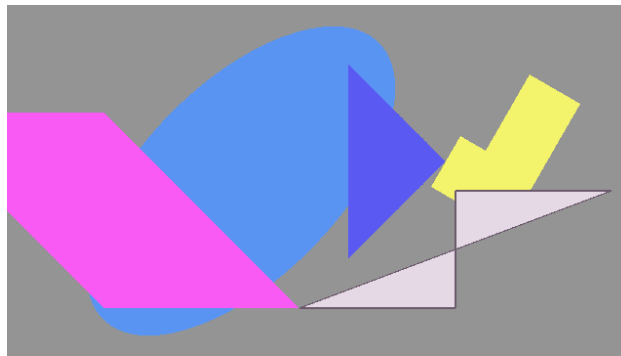


Figure 12: References for Homework 1.4.

5 Antialiasing (10 pts)

If you zoom in to the pictures you generated, you will find that there are some ugly jagged edges around the object boundaries. This phenomenon is called “aliasing” in signal processing literature (we will discuss this in depth in CSE 168). The solution to aliasing is called antialiasing. We are going to implement the simplest form of antialiasing called “supersampling”. The idea is simple: instead of only testing/sampling the pixel center, we’ll sample multiple points within a pixel, and take average between them. For this homework, we will do a 4×4 pattern: we subdivide a pixel into 16 subpixels, compute the color at the center of the subpixels, and take an average over them.

Go to `hw_1_5` in `hw1.cpp` and implement the antialiasing scheme. Test your rendering using the following commands:

```
./balboa -hw 1_5 ../scenes/hw1/antialiasing.json
```

We show our rendering of the `antialiasing` scene in Fig. 13. We will grade your results using other scenes.

We will grade your results using the following scenes:

```
./balboa -hw 1_5 ../scenes/hw1/transformation_2.json
./balboa -hw 1_5 ../scenes/hw1/sun.json
./balboa -hw 1_5 ../scenes/hw1/piggy.json
```

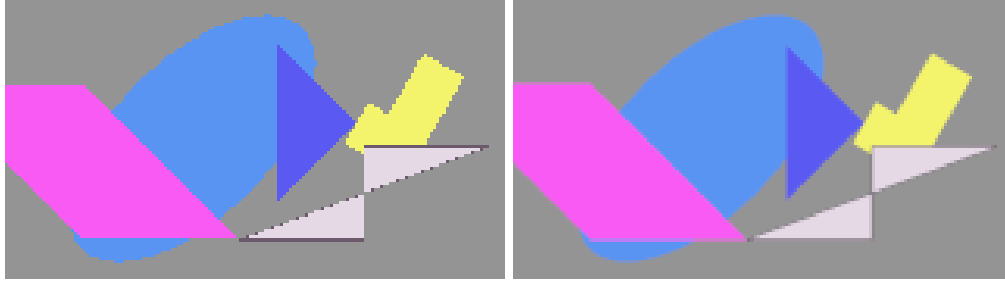


Figure 13: Without/with antialiasing.

In your submission, save your renderings of the scenes above as

```
outputs/hw_1_5_transformation_2.png
outputs/hw_1_5_sun.png
outputs/hw_1_5_piggy.png
```

We will grade by comparing to our rendering.

6 Alpha blending (20 pts)

Finally, let's add some transparency into the objects. In addition to the RGB color, now each object also equips an α (alpha) value. The lower the alpha is, the more transparent the object. $\alpha = 0$ means that the object is fully transparent (so you can't see it), and $\alpha = 1$ means that the object is fully opaque (so it's the same as previous parts of the homework).

For objects with $0 < \alpha < 1$, we need to blend their colors. If there is only one object with alpha α_0 and color C_0 , we blend it with the background color B and obtain the final color C as:

$$C = \alpha_0 C_0 + (1 - \alpha_0) B. \quad (4)$$

What if we have two objects? We can see α as the amount of lights captured at that layer, and $1 - \alpha$ as the amount of light pass through. So for two objects with alpha α_0 , α_1 and color C_0 and C_1 , and object 0 is in front of object 1, we can blend them as:

$$C = \alpha_0 C_0 + \alpha_1 (1 - \alpha_0) C_1 + (1 - \alpha_0)(1 - \alpha_1) B. \quad (5)$$

We'll let you figure out the general formula for arbitrary number of objects and turn it into code. Remember to compose between stroke color and fill color correctly – they are treated as different objects. If you are interested, [here](#) is a great blog post talking about alpha compositing (optional reading). You should still antialias your rendering like the previous part.

Go to `hw_1_6` in `hw1.cpp` and implement the alpha blending. Test your rendering using the following command:

```
./balboa -hw 1_6 ../scenes/hw1/alpha.json
```

We show our rendering of the alpha scene in Fig. 14.

We will grade your results using the following scenes:

```
./balboa -hw 1_6 ../scenes/hw1/alpha_2.json
./balboa -hw 1_6 ../scenes/hw1/alpha_circles.json
./balboa -hw 1_6 ../scenes/hw1/alpha_triangles.json
```

In your submission, save your renderings of the scenes above as

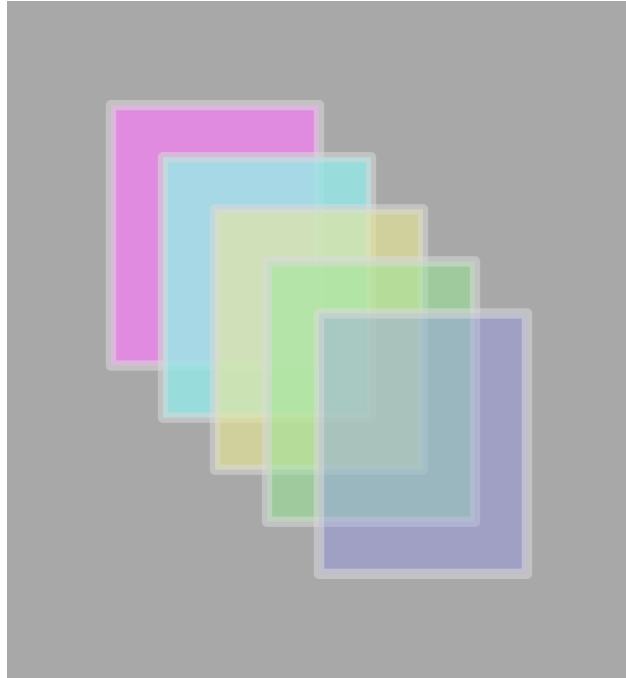


Figure 14: HW 1.6 reference.

```
outputs/hw_1_6_alpha_2.png
outputs/hw_1_6_alpha_cirlces.png
outputs/hw_1_6_alpha_triangles.png
```

7 Design your own scenes (10 pts)

Design a scene yourself and submit the scene and your rendering to us. Be creative! We will give extra credits to people who impress us.

8 Bonus: Bézier curve rendering (15 pts)

Lines and curves are very useful primitives in 2D rendering. As a bonus, implement quadratic Bézier curves as new primitives, and extend the polyline shape to use them. We should have covered what are Bézier curves in the lectures. For the bonuses, feel free to modify any part of the code in balboa.

9 Bonus: Performance optimization (15 pts)

Bruteforcely checking all pixels against all shapes is time consuming when the number of shapes is large. One way to speed up the computation is to compute an axis-aligned *bounding box* for each of the shape, and use it to quickly skip the pixels that do not overlap with the bounding box. For this bonus, create a scene with at least 1000 shapes, render the scene, and try to make it as fast as possible. Report the speedup (and then appreciate how fast your browser is at rendering all the text and UI elements!). `src/timer.h` provides a timer utility you can use for measuring the rendering time.

10 Bonus: Animation (10 pts)

Generate an animation by interpolating between transformations. You can use `ffmpeg` to convert a sequence of images into a video file.