UCSD CSE 167 Assignment 1:
2D Graphics

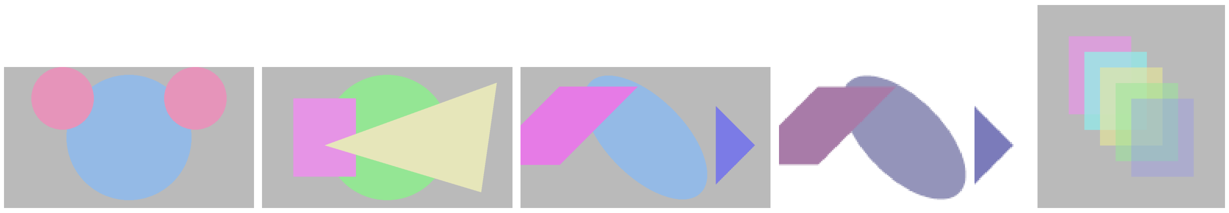


Figure 1: Images we will produce in this homework.

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.

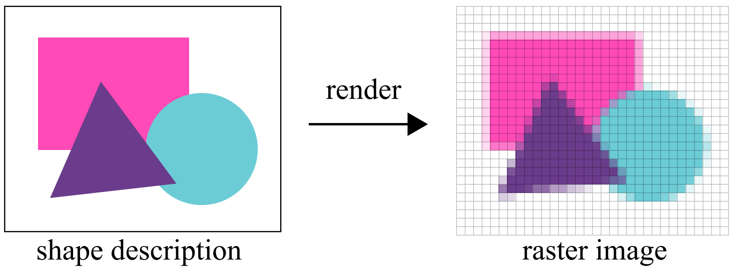


Figure 2: In this homework, we will render a set of 2D shapes into a raster image.

In our first homework, we will implement a 2D renderer that takes a set of simple shapes (circle, 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 convienient 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.

Submission. Submit your code and the outputs of your code through Canvas.

Grading. We will compare your outputs to our reference solutions.

Colloboration 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 (15 pts)

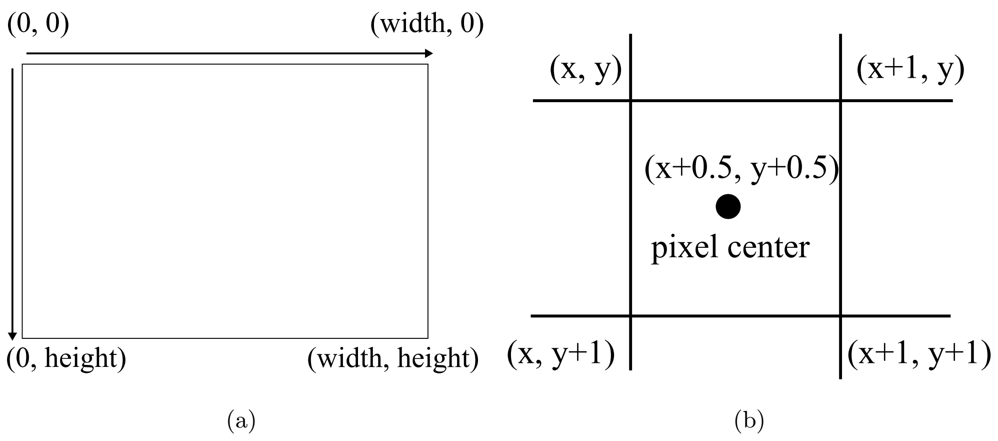


Figure 3: (a) The coordinate system of the "canvas" we will draw our shapes on. (b) The pixel center of for pixel (x, y) locates at $(x + 0.5, y + 0.5)$.

In the first part of this homework, we will render a single circle on our image. We are given a gray "canvas" (Fig. 3a), 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. The usually convention for an image is that the top left is the origin, and the x-axis points towards right, and the y-axis points downwards. We are futher 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 utitilies 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;
```

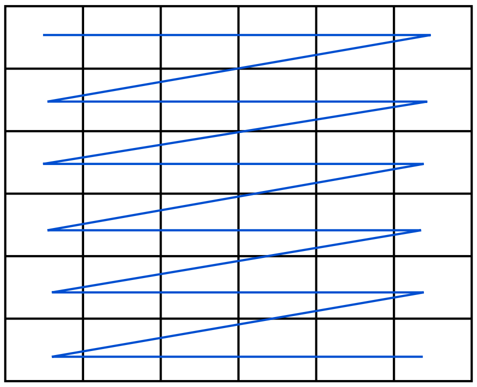


Figure 4: Storing a 2D image in an 1D array by scanning the image.

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 (Fig. 4). 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.

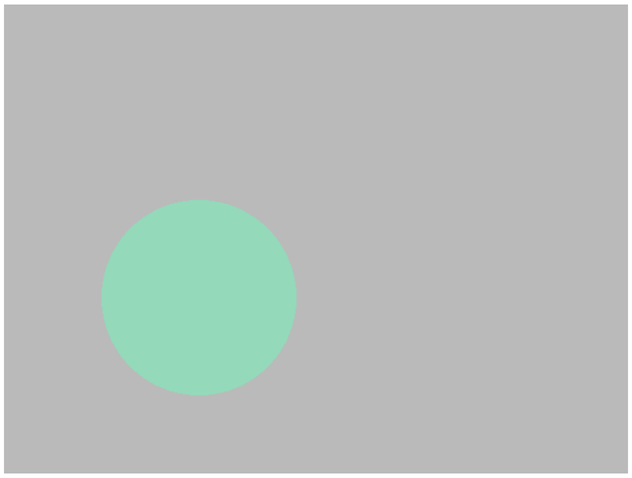


Figure 5: References for Homework 1.1.

If you see your circles, congratulations! If you have not done graphics-related stuff before, this is likely the first time you have 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, 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 multiple circles (15 pts)

In this part, we will extend our previous code to handle multiple circles. This mostly boils down to adding a for loop to check over all of the circles for all of the pixels. However, things become more interesting here: it turns out there are two ways to structure the loops:

```
# For each pixel, check all circles
for each pixel:
    for each circle:
        # check if the pixel center hits the circle
        # overwrite color if the circle is closer
# For each circle, check all pixels
for each circle:
    for each pixel:
        # check if the pixel center hits the circle
        # overwrite color if the circle is closer
```

Think for a bit: do these two produce the same results? The first loop structure finds the closest hitting circle for each pixel, and the second loop structure *paints* each circle on the canvas in order. It will be clearer in the future lectures that in 3D graphics, the first loop structure is usually called *ray tracing* (or ray casting) and the second loop structure is called *rasterization*.

Another small complication we need to resolve is to decide which circle is the closest when there are multiple ones overlapping with a pixel. We simply assume the circles are ordered from the farthest to the closest.

For this part, you can choose to implement either style above that suits you better. The scene is given in the array `hw1_2_scenes` in `hw1_scenes.h`. Note that the scene also specifies the background color and the canvas resolution. We will select the scenes using command line arguments. Implement your rendering code in the function `hw_1_2` and see the result using the following commands:

```
./balboa -hw 1_2 [scene_id]
```

where `[scene_id]` is the scene you want to render (0-4).

We show our rendering for scene 0 in Fig. 6. In your submission, save the images for scenes 1-4 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
```

We'll compare your output with our rendering for grading.

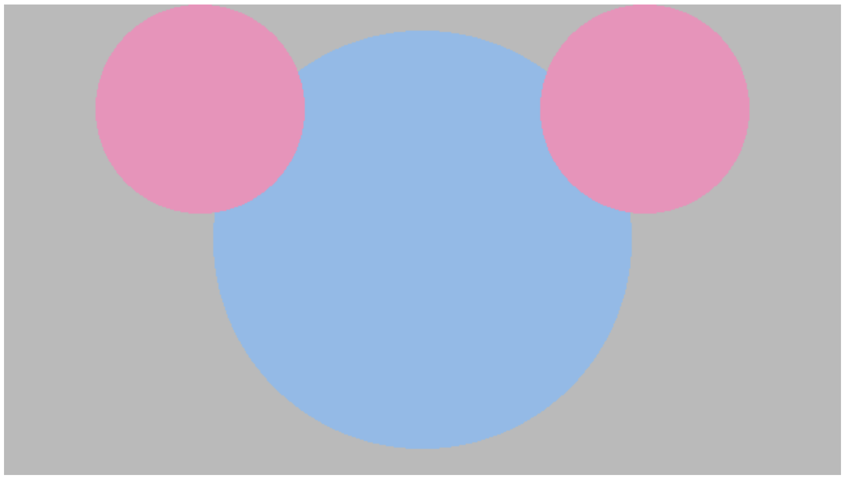


Figure 6: References for Homework 1.2.

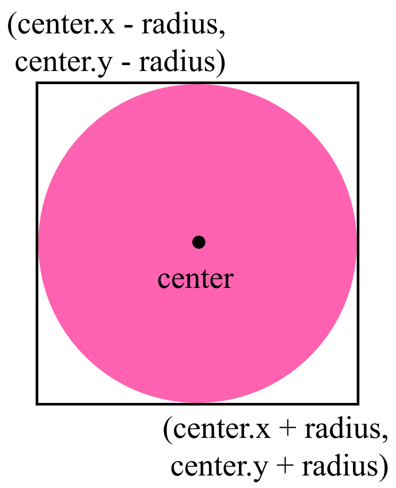


Figure 7: The axis-aligned bounding box of a circle.

Bonus (15 pts): accelerating rasterization using bounding boxes. In practice, we may have a lot of circles and a lot of pixels. So we want to pick a way to render them that is the fastest. Both loop structures can be sped up significantly. The observation is that most of the time, each pixel would only hit a small number of circles. Or equivalently, each circle only will overlap with a small amount of pixels. It turns out that the acceleration for the first loop structure (ray tracing) is a bit more involved (we will discuss them in CSE 168), so we will only talk about the acceleration for the rasterization part.

Notice that each circle usually only intersect with a small set of pixels. Therefore, we can modify the rasterization loop into the following:

```
# For each circle, check all pixels
for each circle:
    bounding_box = \
        BBox(circle.center - radius,
             circle.center + radius)
    for each pixel in bounding_box:
        # check if the pixel center hits the circle
        # overwrite color if the circle is closer
```

Here, for each circle, we can compute an *axis-aligned bounding box* that contains the circle (Fig. 7), and only loop over the pixels that overlaps with the bounding box. An axis-aligned bounding box is defined by two points: the point on the top left, and the point at the bottom right.

For the bonus, implement the rasterization loop above, and report the speedup you obtain before and after the optimization.

3 Rendering more shapes (15 pts)

Next, we will add more types of shapes into our renderer. In particular, we will support rendering an (axis-aligned) rectangle and a triangle. Before that, let's introduce our scene file format, so that we don't have to hardcode our scenes in a C++ header. 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 showing scene 0 from the previous part in our scene format:

```
{
    "resolution": [640,360],
    "background": [
        0.5, 0.5, 0.5
    ],
    "objects": [
        {
            "type": "circle",
            "center": [320,180],
            "radius": 160,
            "color": [0.3, 0.5, 0.8]
        },
        {
            "type": "circle",
            "center": [150,80],
            "radius": 80,
            "color": [0.8, 0.3, 0.5]
        },
        {
            "type": "circle",
            "center": [490,80],
            "radius": 80,
            "color": [0.8, 0.3, 0.5]
        }
```

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

```
    ]
}
```

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

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

where `Shape` is a `std::variant`:

```
struct Circle {
    Vector2 center;
    Real radius;
    Vector3 color;
    Matrix3x3 transform;
};

struct Rectangle {
    Vector2 p_min, p_max;
    Vector3 color;
    Matrix3x3 transform;
};

struct Triangle {
    Vector2 p0, p1, p2;
    Vector3 color;
    Matrix3x3 transform;
};

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

Don't worry about `transform` 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;
    // ...
  }
};
```

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 *rectangle = std::get_if<Rectangle>(&shape)) {
    // do something with rectangle
} else if (auto *triangle = std::get_if<Triangle>(&shape)) {
    // do something with triangle
}
```

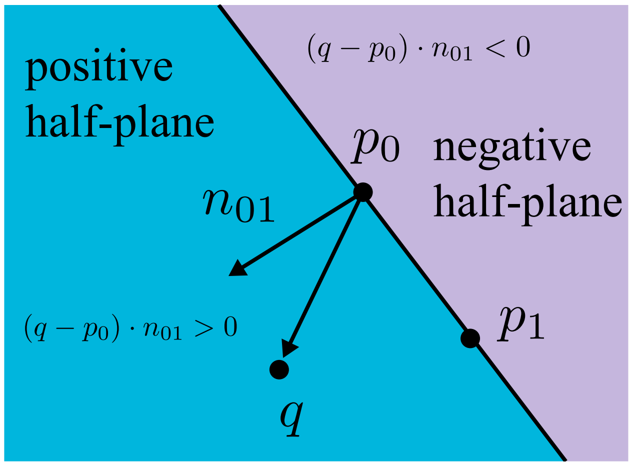


Figure 8: Each edge of a triangle forms a line that splits the 2D space into two half planes. Here we show the edge from p_0 and p_1. The line has a "normal" direction n_{01} that is perpendicular to the edge direction (i.e., $n_{01} \cdot (p_1 - p_0) = 0$). For a point q, we can determine which half-plane it is in by forming a vector with p_0 and taking dot product with n_{01}. If the dot product is positive, it is on the same side as where the normal is pointing at, and vice versa.

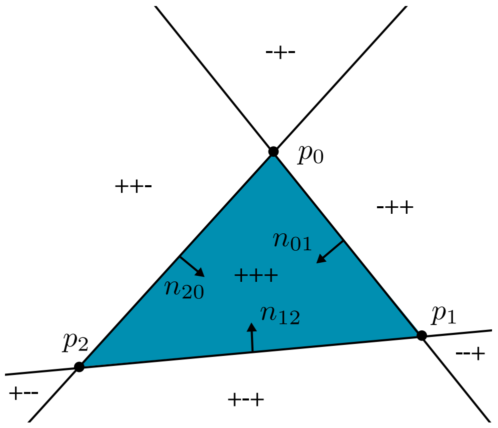


Figure 9: The 6 half-planes formed by the 3 edges split the 2D plane into 7 regions. If all normals are pointing inwards, then the interior of the triangle is the intersection of all three positive half-planes. Otherwise the interior is the intersection of all negative half-planes.

A `Rectangle` is axis-aligned and is defined by its top-left (`p_min`) and bottom-right (`p_max`) points. Testing whether a 2D point is inside an axis-aligned rectangle is easy and I'll leave it to you to figure it out.

A `Triangle` is defined by three points (`p0, p1` and `p2`). Testing whether a point is inside a triangle is a bit more involved. There are more than one ways to do it, but I like to do it by looking at the *half-planes* formed by the three edges of a triangle. For each edge of the triangle, we can form a line that splits the whole 2D space into two parts, we call these parts half-planes (Fig. 8). We can determine whether a point q is on the positive or negative half-planes by looking at the sign of the dot product between a vector formed by the point q and the line, and a normal vector n that is perpendicular to the direction of the line.

The key observation for testing whether a point is inside a triangle or test, is to notice that if we follow the three edge directions: $p_1 - p_0$, $p_2 - p_1$, and $p_0 - p_2$, and rotate them by 90 degrees in clockwise direction to obtain the normals n_{01}, n_{12}, and n_{20}, then the interior of the triangle is the intersection of all the positive half-planes or all the negative half-planes. More precisely, if p_0, p_1 and p_2 follows the clockwise direction, then the interior would be the intersection of all the positive half-planes, otherwise if p_0, p_1 and p_2 follows the counterclockwise direction, then the interior would be the intersection of all the negative half-planes (Fig. 9).

The remaining question is how do we rotate the edge directions to obtain the normals? Recall that the 2D counterclockwise rotation matrix is:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \tag{1}$$

where θ is the (counterclockwise) rotation angle – plugging in $\theta = -90°$ we obtain:

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ -x \end{bmatrix}. \tag{2}$$

Thus, to obtain normal n_{01}, we can take the edge $e_{01} = p_1 - p_0$, and set $n_{01}.x = e_{01}.y$ and $n_{01}.y = -e_{01}.x$.

At this point, you should have enough knowledge to implement a renderer that can handle circles, rectangles, and triangles. Write your code in `hw_1_3` and test it using the following commands:

```
./balboa -hw 1_3 ../scenes/hw1/three_circles.json
```

```
./balboa -hw 1_3 ../scenes/hw1/three_shapes.json
```

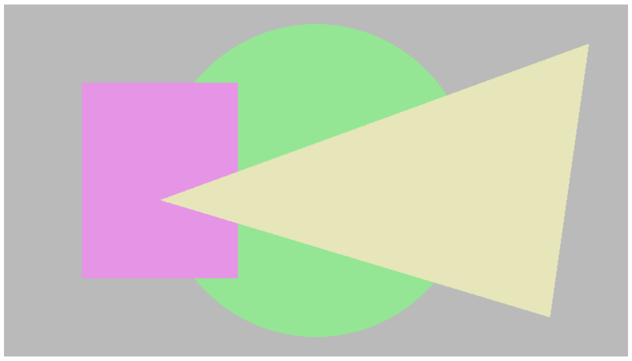


Figure 10: References for Homework 1.3.

We show our rendering of the `three_shapes` scene in Fig. 10. The `three_circles` scene should be the same as Fig. 6. We will grade your results using the following scenes:

```
./balboa -hw 1_3 ../scenes/hw1/three_shapes_2.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_three_shapes_2.png
outputs/hw_1_3_cat.png
outputs/hw_1_3_robot.png
```

We will grade by comparing to our rendering.

4 2D transformation (20 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.

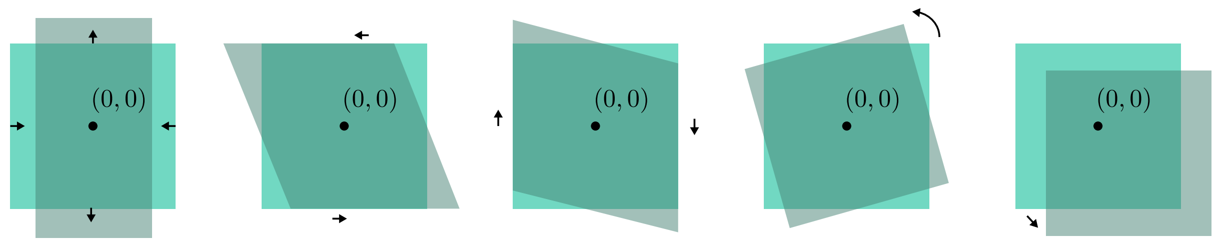
In this part, we will implement a few common affine transformations: scaling, shearing, rotation, and translation. Crucially, we also want to *combine* these transformations: for example, we might want to first rotate the shape, then translate it. As noted in Fig. 11, everything except for translation can be represented as a matric 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

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & b_0 \\ a_{10} & a_{11} & b_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3}$$

is equivalent to

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \tag{4}$$

Therefore, we can represent all affine transformations using the 3×3 matrix above. Fig. 12 shows the corresponding version of the matrices for each transformation.



$$\begin{bmatrix}x'\\y'\end{bmatrix} = \begin{bmatrix}s_x & 0\\ 0 & s_y\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

(a) scale

$$\begin{bmatrix}x'\\y'\end{bmatrix} = \begin{bmatrix}1 & \lambda_x\\ 0 & 1\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

(b) shear x

$$\begin{bmatrix}x'\\y'\end{bmatrix} = \begin{bmatrix}1 & 0\\ \lambda_y & 1\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

(c) shear y

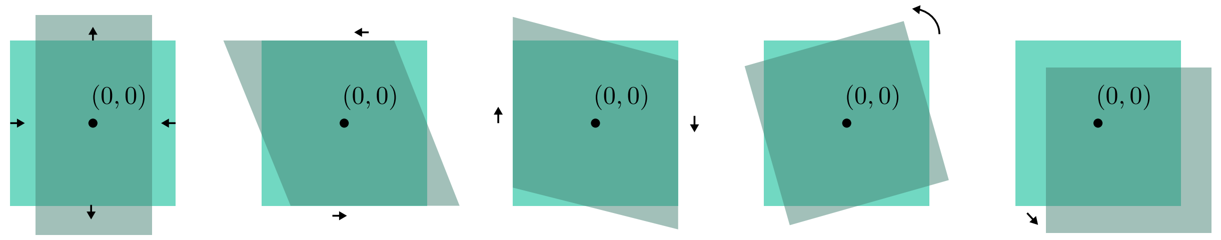
$$\begin{bmatrix}x'\\y'\end{bmatrix} = \begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

(d) rotate

$$\begin{bmatrix}x'\\y'\end{bmatrix} = \begin{bmatrix}x\\y\end{bmatrix} + \begin{bmatrix}t_x\\t_y\end{bmatrix}$$

(e) translate

Figure 11: 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.



$$\begin{bmatrix}x'\\y'\\1\end{bmatrix} = \begin{bmatrix}s_x & 0 & 0\\ 0 & s_y & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix}$$

(a) scale

$$\begin{bmatrix}x'\\y'\\1\end{bmatrix} = \begin{bmatrix}1 & \lambda_x & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix}$$

(b) shear x

$$\begin{bmatrix}x'\\y'\\1\end{bmatrix} = \begin{bmatrix}1 & 0 & 0\\ \lambda_y & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix}$$

(c) shear y

$$\begin{bmatrix}x'\\y'\\1\end{bmatrix} = \begin{bmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix}$$

(d) rotate

$$\begin{bmatrix}x'\\y'\\1\end{bmatrix} = \begin{bmatrix}1 & 0 & t_x\\ 0 & 1 & t_y\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix}$$

(e) translate

Figure 12: 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. 11.

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

The circle has the default radius (1) and center ((0, 0)). The transformations are applies 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:

$$\text{translate}(\text{rotate}(\text{scale}(x))) = TRSx = Fx \tag{5}$$

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

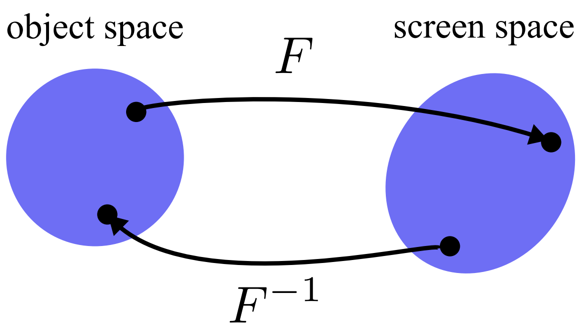


Figure 13: Transformation converts points between spaces.

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 tranformation (let's call it F) converts points in the object space to the *screen space*. On the other hand, the *inverse* of the transformation F^{-1} (i.e., matrix inversion) converts points in the screen space to the object space. See Fig. 13 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 screen space. Given a screen 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. 14.

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.

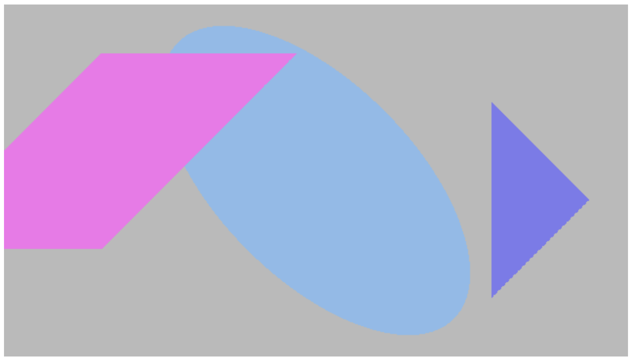


Figure 14: References for Homework 1.4.

Bonus (15 pts). Generate an animation by interpolating between transformations. Feel free to modify any part of the code in balboa. You can use ffmpeg to convert a sequence of images into a video file.

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 phenomonon 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. 15. 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
```

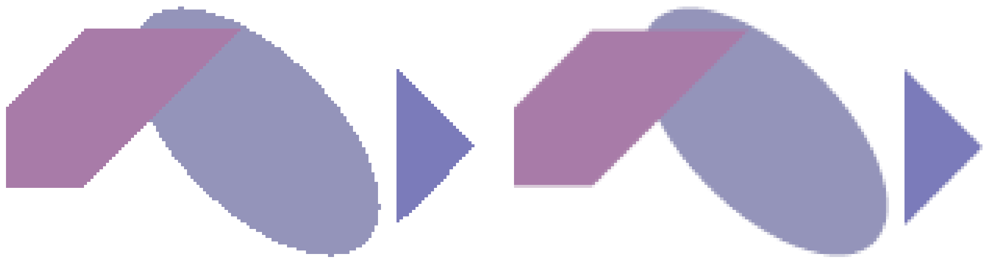


Figure 15: Without/with antialiasing.

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

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 (15 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:

$$C = \alpha_0 C_0 + (1 - \alpha_0) B. \tag{6}$$

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 infront of object 1, we can blend them as:

$$C = \alpha_0 C_0 + \alpha_1 (1 - \alpha_0) C_1 + (1 - \alpha_0)(1 - \alpha_1) B. \tag{7}$$

We'll let you figure out the general formula for arbitrary number of objects and turn it into code.

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. 16.

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

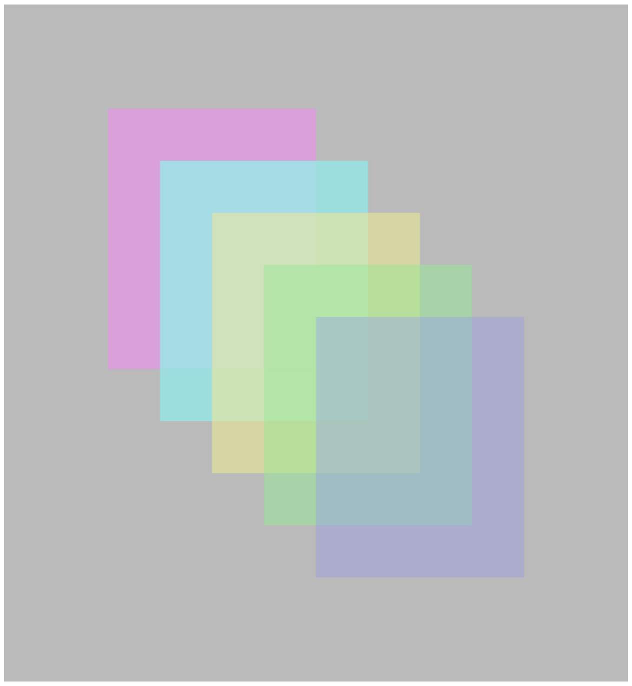


Figure 16: HW 1.6 reference.

```
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: line and Bézier curve rendering (15 pts)

Lines and curves are very useful primitives in 2D rendering. As a bonus, implement lines, quadratic Bézier curves, and cubic Bézier curves as new primitives. A line takes two points and a width (basically a rotated rectangle), a quadratic Bézier curve takes three control points and a width, and a cubic Bézier curve takes four control points and a width. Read A Primer on Bézier Curves for the definition of Bézier curves and how to convert them to lines. Alternatively, you can directly compute the distance between a point and a Bézier curve: for quadratic curves, this boils down to a cubic equation root solve, and for cubic Bézier curve, this becomes a quintic equation root solve, and requires an iterative solver.