Comp360/560 Lab 2 : Iterated Function Systems

Due Date: Feb 23, 2022, 11:55pm

1 Overview

In this project you will create fractals using iterated function systems (IFS) as shown in class. This project has three parts that you will need to complete. First, you will write code to create new polygonal shapes. Second, you will complete the definition of a matrix class, and use that class to define affine transformations. Finally, you will use a graphical interface to construct IFS by creating and transforming triangles.

This project is due on Feb 23, 2022 at 11:55pm. It is worth 100 points.

You should work in groups of two. You may not work with the same partner as you did in Lab 1.

2 Program Overview

The given code can be compiled directly without change. The window in **Figure 1** should appear when compilation finishes. This program has two display boxes: the left black box is the **Geometry Viewer**, and the right black box is the **IFS Viewer**. With the **Geometry Viewer**, you will be able to create and manipulate basic polygonal shapes, and these shapes will serve as the base cases for the fractals. In the **IFS Viewer** you will create transformations, which are represented as triangles. **IFS Viewer** allows you to transform the triangles through mouse manipulations or through specifying an affine transformation. The bottom panel of the program contains several buttons, a slider, and a list. These UI elements will be further specified below.

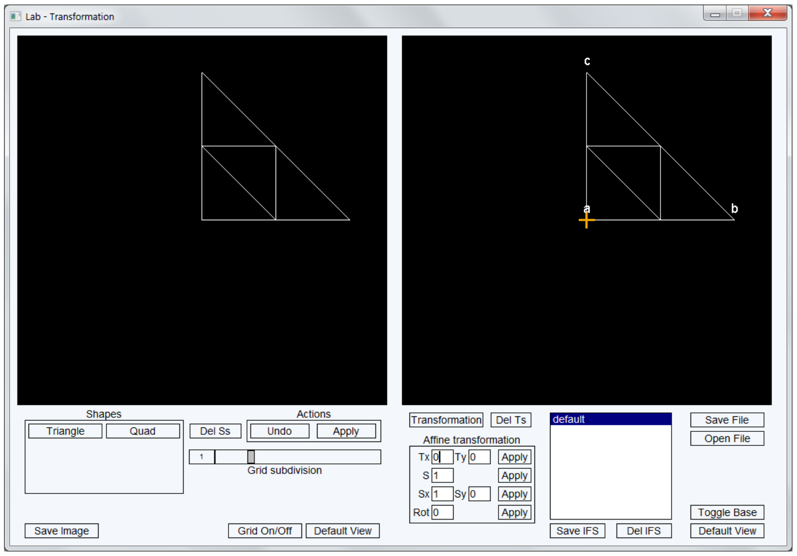

Figure 1: after compilation

2.1 Mouse operations

The two viewers have an identical set of operations that can be performed through the mouse.

Moving the mouse

Moving the mouse without clicking or dragging a button has no effect on the content of the displays. However, if the mouse hovers over a polygonal shape, then that polygonal shape will be highlighted with a red border to indicate that the polygon can be selected or edited. Similarly, if the mouse hovers over a corner of a polygonal shape or the center of transformation (orange cross), then that point will be highlighted in red.

Left mouse button

- Left click on a highlighted point/shape will select the point/shape.
- Dragging a selected point/shape moves the point/shape with respect to the mouse.
- Left click and drag on empty space in the display will pan the viewing window.

Right mouse button

- Right click on a highlighted shape will switch the shape into an editing state (see Sections 2.2.1 & 2.2.3).
- Right click and drag on empty space in the display will zoom the viewing window.

2.2 Panel Operations

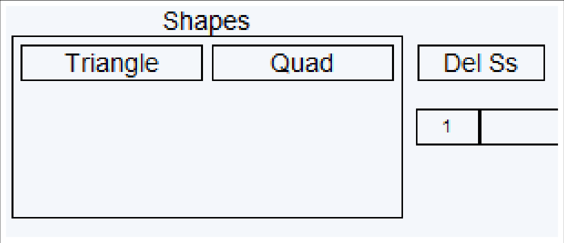

Figure 2: Shape controls

2.2.1 Shapes

In the "Shapes" frame (Figure 2), there are currently two buttons "Triangle" and "Quad". Pressing one of the two buttons will create a triangle or quad in the **Geometry Viewer**. One of your tasks is to add more shapes to the "Shapes" frame (see Section 3.2). Pressing "Del Ss" will delete any shapes that are in the editing state (shapes that are highlighted in green).

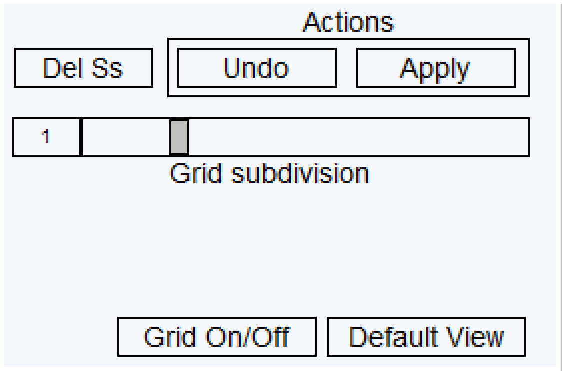

Figure 3: Geometry Viewer options and IFS application

2.2.2 Geometry Viewer options and IFS application

In Figure 3, the "Actions" frame contains the "Undo" and "Apply" buttons. The "Apply" button will apply the current IFS (displayed in the **IFS Viewer**) to the shapes in the **Geometry Viewer**. The "Undo" button will undo the last IFS application.

The "Grid subdivision" bar controls the grid subdivision of the base triangle. Changing the subdivision level has two effects other than the obvious visual change. First, creating a new transformation in the **IFS Viewer** will be scaled accordingly. For example, if the subdivision level is 1, then the size of the new transformation is 1/4 of the base triangle. Second, moving transformations in the **IFS Viewer** snaps to the grid points of the subdivided base triangle.

"Grid On/Off" toggles the display of the base triangle in the **Geometry Viewer**. "Default View" will return the viewing space of the **Geometry Viewer** to its initial state.

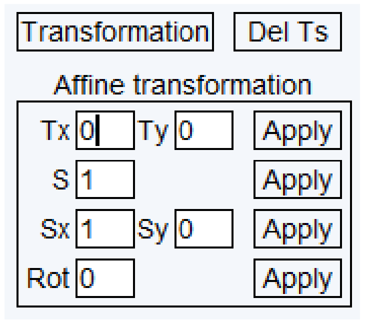

Figure 4: Transformation

2.2.3 Transformations

Pressing the "Transformation" button will place a new transformation in the **IFS Viewer**. Transformations are represented as triangles. The size of the triangle depends on the subdivision level of the base triangle. "Del Ts" will remove any transformation that are in the editing state.

The parameters in the "Affine transformation" group allows you to define an affine transformation that can be applied to any "editing" triangles in the **IFS Viewer**. For translation, you can specify the translation vector in the "Tx" and "Ty" input boxes. For uniform scaling, specify the scaling factor in the "S" input box. For non-uniform scaling (see Chapter 4.7.2), specify the vector of the non-uniform scaling in the "Ux" and "Uy" boxes. For rotation, specify the angle of rotation (in degrees) in the "Rot" box. Uniform scaling, non-uniform scaling, and rotation are defined with respect to a center of transformation. This center is shown as the **the orange cross in the IFS Viewer**. In all cases, pressing the "Apply" buttons will apply the corresponding transformation to any triangles in the editing state.

Please note that these transformations will only affect triangles in the IFS Viewer. We intend you to use these transformation functionalities to define precise IFSs.

2.2.4 IFS operations and others

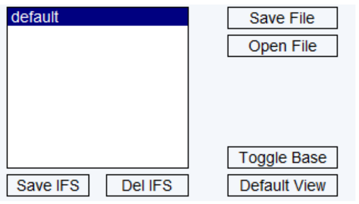

Figure 5: IFS operations and others

The list that contains a "default" entry is the current list of IFSs created by you. Pressing "Save IFS" will bring up a dialog to query for a name. Once a name has been entered, the current IFS will be duplicated and saved. You can switch between the saved IFSs by clicking on the corresponding entries in the list. Entries in the list can be removed by pressing "Del IFS". If the list contains only one entry, that entry cannot be removed.

The IFSs can be saved to a file by using the "Save File" button, which opens a file dialog. Similarly, the "Open File" button will allow you to load IFSs that are stored in a file format. Note that loading a file will clear all previous IFSs.

"Toggle Base" allows you to edit the base triangle in the "IFS Viewer" by moving the vertices of the base. During this edit mode, transformation-triangles cannot be manipulated. Pressing "Default View" will return the viewing space of the **IFS Viewer** to its initial state.

3 Implementation

So far we have only described the exisiting functionalities of this program. This section will describe the requirements that you will need to complete for this project. The implementations can be divided into three parts: create additional shapes, implement a matrix class and apply the class to transformation, and make your own fractals. Before going into the requirements, we will first describe the organization of the given code.

3.1 Notes on the given code

The given code is in C++. The structure is similar to that of Lab 1. Please make sure you have read this section before attempting your implementation.

3.1.1 Class descriptions

First, the following files and their corresponding classes do not require additions or changes: `Button.h`, `Common.h/cpp`, `FrameWindow.h/cpp`, `Manager.h/cpp`, and `BaseGrid.h/cpp`. However, we will not discourage you from exploring and improving the given code. If at any point of your implementation you feel the need to modify any of these classes, please feel free to do so, but please also document your changes in the README.

The GeometryViewer.h/cpp and IFSViewer.h/cpp contain respectively the core code for the **Geometry Viewer** and the **IFS Viewer**. These two classes contain very similar code for handling user interactions. In `GeometryViewer.cpp` you will need to consider the `handle` and `addShapeCb` functions. Read over the code for both functions and make sure you understand the general idea; these two functions will need to be modified to accommodate more shapes.

The `Matrix.h` file contains code for defining NxN matrices and vectors of arbitrary size. The existing code is mostly straightforward. If you have not had much experiences with C++ coding, then reading over materials on operating overloading should be very helpful (additional ref). `Transformation.h` contains the class `Transformation`, which represents an affine transformation as a 3x3 matrix `_mat`. In this class you will need to complete code that modifies a 3x3 matrix with respect to a given transformation (rotation, uniform scaling, non-uniform scaling, or translation). See Section 3.3.2 for more details.

The `TinyGeom.h/cpp` files contain definitions for different shapes such as the Triangle and Quad. You will need to modify this file to add more shapes. This project uses the visitor pattern to allow us to specialize an algorithm for a particular shape. For our purposes, the `Transformation` class is a visitor defined over the shapes. To use the visitor pattern, all shapes should subclass the `Geom2` class. Additionally, the `Geom2Visitor` class also needs to be extended to accommodate more shapes.

3.1.2 Defining vectors and points

We define affine transformation as a composition of rotation, scaling, and translation. Numerically, an affine transformation is represented as a 3x3 matrix (for the reason why, please review the textbook). This matrix representation implies that we need to embed our 2D primitives in 3D. In 2D linear space, we do not have a clear distinction between points and vectors; both are represented as a tuple of scalar values. However, in the 2D affine space, 2D points are represented as (px,py,1) and 2D vectors are represented as (vx,vy,0).

In terms of the given code, the class `Vec2` and `Pt2` are used to define 2D vectors and 2D points. You need to be careful when working with vectors and points; do not use points where you mean vectors and vice-versa. The compiler will not raise errors if you confuse the two, but your program might produce unexpected and erronesou results. You should never use the default constructor to construct a vector or a point. So do not write lines like `Vec2 v;` or `Pt2 p;`. You should always specify the content by writing `Vec2 v(0,0,0)` or `Pt2 p(0,0)`. The last scalar for a `Pt2` instance can be ignored since it defaults to 1. You can also use syntax like `v = Vec2(0,0,0)` and `p = Pt2(0,0)`.

By working out a few simple examples using this representation, we can observe that affine transformations map points to points and vectors to vectors. At no point in your program should you ever observe a non-zero and non-one value as the third scalar for either a vector or a point. This invariant a good sanity check for debugging.

3.2 Additional shapes (16 Points)

We have already implemented "Triangle" and "Quad" for you. Your job is to add a few more shapes and to build an interface for the user to manipulate these shapes in the **Geometry Viewer**. This part will be completed in C++. The requirements are listed below.

Add hexagon, octagon, circle, and one additional shape of your choice

This addition requires several changes to multiple source files. Below are details of the changes you need to make. We recommend you try adding one shape first and then move on to other shapes.

- Subclass Geom2 for your new shape. (`TinyGeom.h`). You should read the code for Triangle and Quad to get a sense of what needs to be done.
- See the TODO's in `TinyGeom.h` on the additions you need to make for the visitor and the shape enumerations. Also add a `visit` function for your new shape in `Transformation.h`.
- Add a button for your new shape. Remember to set the callback function (see `main.cpp`).
- Extend `addShapeCb` and `handle` to account for more shapes (see `GeometryViewer.cpp`).
    - If the shape is convex (triangle, quad, hexagon, octagon), mouse manipulation of your shape should maintain the convexity of the shape.
    - An easy way to main convexity is to let translation of the selected vertex represent a uniform scaling from the centroid of the shape. You can also specialize your manipulation as was done for the Quad.
    - For the circle, moving a vertex on the circumference should correspond to changing the radius of the circle.
    - We only require that manipulation is well-defined for first-level primitives (i.e. before any IFS has been applied to the shapes).
- The one additional shape can be any shape, but note that the given code only handles mouse manipulation of convex shapes. **Please document your new shape in the README**. Add some details about the definition and any special handling code you write for this particular shape.

After completing the above requirements, your program should be able to add new shapes.

3.3 Matrix and transformation

3.3.1 Matrix definition (10 Points)

We have provided the skeleton of the `Matrix.h` class. You need to fill in three operators defined on `Matrix`: `operator+`, `operator*`, and `operator!`. `operator+` adds two matrices of the same size, `operator*` multiplies two matrices, and `operator!` inverts a square matrix. For our purposes, you can assume the matrix is always of size 3x3. In the case of a singular matrix, output an error message and return the input matrix.

Matrix multiplication and inversion are the only operations we will need for this project. We will check the code for adding matrices for correctness, but we will not test the code during program execution.

3.3.2 Transformations (15 Points)

The `Transformation` class in `Transformation.h` encapsulates affine transformations, which are represented as 3x3 matrices. Your task is to complete the following transformations (their C++ function names are in parenthesis):

- **Translation**: translates a point or shape by an input vector (`setAsTranslate` and `composeTranslate`).
- **Uniform scaling**: scales a point or shape about the center of transformation by a scaling factor (`setAsScale` and `composeScale`).
- **Non-uniform scaling**: scales a point or shape about the center of transformation by a vector. You will use the normalized version of the input vector as the vector of scaling. Also, you will compute the magnitude of the input vector and use the result as the scale factor(`setAsNUScale` and `composeNUScale`). Also see Chapter 4.7.2 in the textbook.
- **Rotation**: rotates a point or shape about the center of transformation by an angle (in degrees) (`setAsRotate` and `composeRotate`).
- **Three-point mapping**: a transformation that takes three source points to three destination points (`setAs3PtTransform` and `compose3PtTransform`). The points are given as two triangles.

For any transformation that depends on a center of transformation, this center is displayed as an orange cross in the **IFS Viewer**. You may move the center around just as you would the corners of a triangle. The `setAs..` functions initialize the transformation to the identity mapping and apply the respective affine transform to the identity map. The `compose...` functions compose affine transformations. Consider if we write the following sequence of code for `Transform t` and `Transform u`

- `t.setAsTranslate(Vec2(1,1,0));`
- `u.setAsTranslate(Vec2(1,1,0));`
- `u.composeRotate(45,Pt2(0,0));`

`t` is a transformation that translates a point by the vector (1,1). On the other hand, `u` is a transformation that first rotates the input point about the origin by 45 degrees and then translates the point by the vector (1,1). Note that calling a `compose..` function is equivalent to a **left multiplication** to the transformation matrix by another transformation. This means that even though the `composeRotate` call comes after `setAsTranslate`, rotation is the first transformation that is applied to the input point, and translation comes after rotation. This convention is used in OpenGL and many other major graphics packages.

You should also enforce the convention of multiplying the input on the **right side** by the transformation matrix. This notation is the same convention as in the textbook. Following this convention allows you to better connect your code with the notation in the textbook. You can find details on how to implement affine transformations in the course textbook.

3.4 IFS, simple operators, and fractals

Once you have completed the requirements of Seciton 3.3, you should be able to apply iterated function systems to shapes in the **Geometry Viewer**. In this given framework, all IFSs are defined using one or more mappings of three points. A transformation is defined in terms of a base triangle (outlined in white in regular mode, red in base edit mode) and a collection of destination triangles. Each destination triangle corresponds to a mapping from the base triangle (the source) to the destination triangle. In the **IFS Viewer**, you can add destination triangles (or transformations) by pressing the "Transformation" button. The size of the new triangle depends on the current subdivision level of the base triangle.

You should be able to easily create a Sierpinski Gasket by following these steps: create three destination triangles, each of size 1/4 of the base triangle; let them snap to the corners of the base triangle, and press "Apply" in the "Actions" group. Completing these steps should generate a first level Sierpinski Gasket.

3.4.1 Simple operators (10 points)

You need to complete the following simple operators. These operators are similar to what you have completed in the turtle lab. For this project, you will be required to use fixed values.

- **Spin**: Construct 10 rotated version of a given shape. Each copy is a 36 degree rotation from the previous copy (except for the first copy).
- **Shift**: Construct 10 translated copies of a given shape. Each copy should be a translated version of the previous copy. Translate each copy by the vector (20,20).
- **Scale**: Construct 10 scaled copies of a given shape. Each copy should be a scaled version of the previous copy. Scale by the scalars (1.2,.8) in the x and y directions, respectively.

We do not recommend constructing these operators by using the mouse. By using the options in the "Affine Transformation" group, you should be able to construct these operators exactly. **Save these operators (IFSs) into a file called "SimpleOps.txt".**

3.4.2 Fractals (25 points)

You need to construct the following fractals. Many of these fractals you have constructed during the turtle graphics lab; some of these fractals can be found in the textbook.

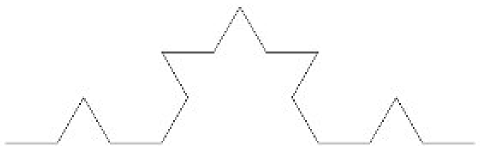
Figure 6: KochSnowflakeSide at recursion level 2

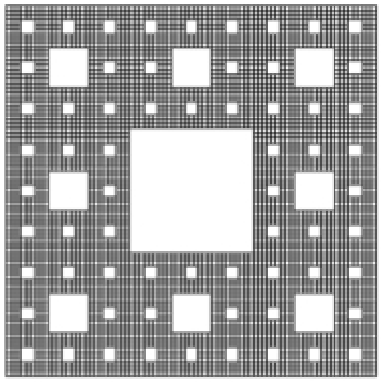
Figure 7: SquareGasket

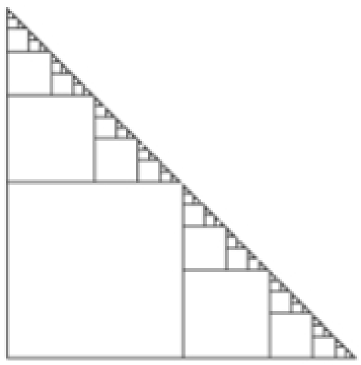
Figure 8: FractalStairs

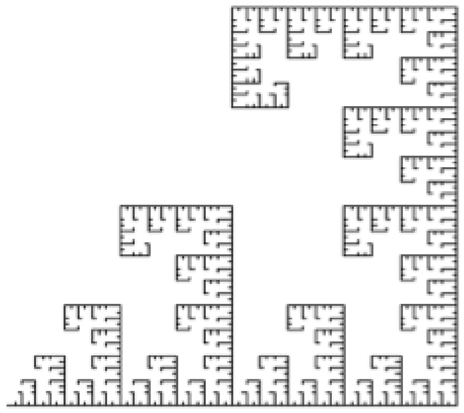
Figure 9: FractalHangman

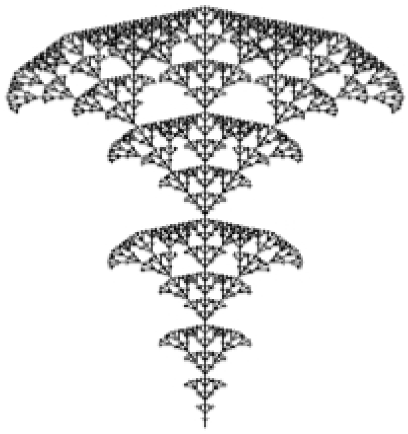
Figure 10: Bush

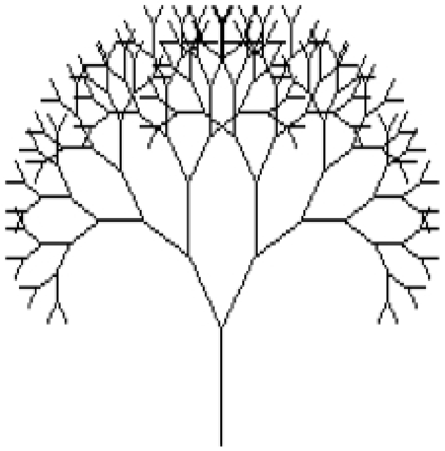
Figure 11: Tree

- **KochSnowflakeSide**: see Figure 6.
- **SnowflakeSide4**: Draws a bump fractal containing a polygon with 4 sides. This fractal is the generalization of the Koch curve to arbitrary polygons (refer to the turtle graphics lab).
- **SnowflakeSide5**: Same as above but with 5 sides.
- **Snowflake**: create a three-sided snow-flake where each side is a KochSnowFlakeSide
- **SquareGasket**: see Figure 7.
- **FractalStairs**: see Figure 8.
- **FractalHangman**: see Figure 9.
- **FractalBush**: see Figure 10.
- **FractalTree**: see Figure 11 You can assume that we will use a vertical line as the base case.
- **FractalC**: Construct the fractal C-Curve illustrated in Chapter 2, Figure 1.

Save these fractals into a single file call "BasicFractals.txt". This file should reside in the "files" folder in the project directory.

3.4.3 Four self-created fractals with pictures (24 points)

You need to create four fractals of your own design. These fractals should be different from the fractals you constructed in the previous section. Please put in some effort to create interesting fractals; part of your grade will be based on the artistic merit of your fractal. **Save your fractals as files in the "files" directory**. You may save one or multiple fractals per file. Please document these fractals (their file names and any other noteworthy details)in the README. **Please also capture an image for each of your fractals** using the "Save Image" button at the bottom left-hand corner of the program panel. Put the images in the "images" folder.

4 What to turn in

Use the given "Lab2" directory. This directory should contain your source code and all files needed to build your program in Visual C++, all text files containing each of your IFS programs (in each file just include a single IFS program), and all the images generated by your IFS program. The grader should be able to load your fractal programs (saved in text files) and test them. Also include a README file containing:

- The names of you and your partner.
- Instructions on running your fractals or other programs.
- Any other implementation details you think your grader should know.

The compressed "Lab2" directory should be submitted to Canvas.