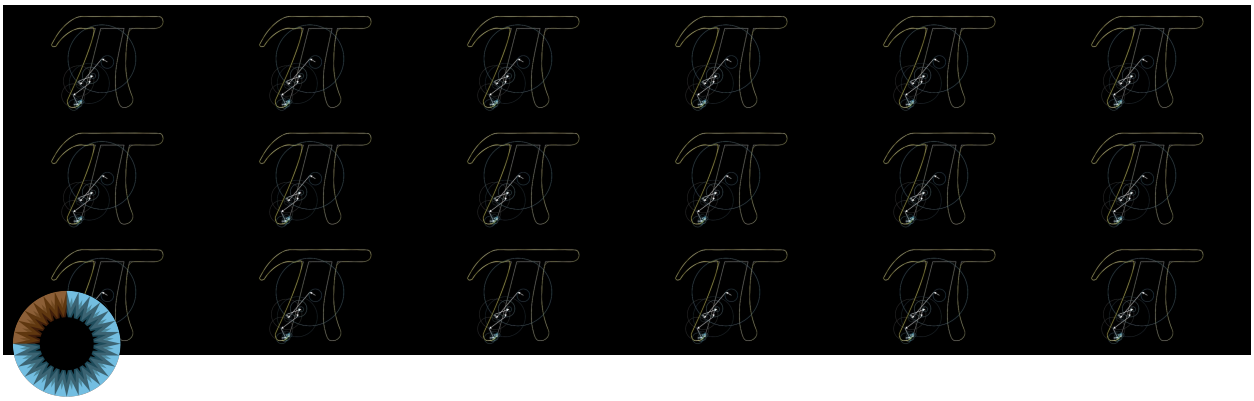

ManimGL for Beginners

by Jonathan Waldorf

Now that we've set up our ManimGL configuration in this guide, let's learn how to use this software to make animations! We'll be going through the example scenes provided by the ManimGL documentation to see what this software is capable of. Although renderings and explanations can be found on the website, these scenes were downloaded on our device when we installed ManimGL. Assuming we were just in our `Animation_Projects` folder, we're going to take a step back by typing `cd ..` and enter the `manim` folder instead with `cd manim`. In here there should be a file called `example_scenes.py` that we'll open in Sublime.

Interactive Scenes

First we're going to learn more about the interactive scenes discussed earlier. Looking at the InteractiveDevelopment class:

```
class InteractiveDevelopment(Scene):
    def construct(self):
        circle = Circle()
        circle.set_fill(BLUE, opacity=0.5)
```

```
        circle.set_stroke(BLUE_E, width=4)
        square = Square()

        self.play(ShowCreation(square))
        self.wait()

        # This opens an iPython terminal where you can keep writing
        # lines as if they were part of this construct method.
        # In particular, 'square', 'circle' and 'self' will all be
        # part of the local namespace in that terminal.
        self.embed()

        # Try copying and pasting some of the lines below into
        # the interactive shell
        self.play(ReplacementTransform(square, circle))
        self.wait()
        self.play(circle.animate.stretch(4, 0))
        self.play(Rotate(circle, 90 * DEG))
        self.play(circle.animate.shift(2 * RIGHT).scale(0.25))

        text = Text("""
            In general, using the interactive shell
            is very helpful when developing new scenes
        """)
        self.play(Write(text))

        # In the interactive shell, you can just type
        # play, add, remove, clear, wait, save_state and restore,
        # instead of self.play, self.add, self.remove, etc.

        # To interact with the window, type touch().  You can then
        # scroll in the window, or zoom by holding down 'z' while scrolling,
        # and change camera perspective by holding down 'd' while moving
        # the mouse.  Press 'r' to reset to the standard camera position.
        # Press 'q' to stop interacting with the window and go back to
        # typing new commands into the shell.
```

```
# In principle you can customize a scene to be responsive to
# mouse and keyboard interactions
always(circle.move_to, self.mouse_point)
```

We can render this scene using `manimgl example_scenes.py InteractiveDevelopment` . At first glance, it seems that this code would move the code around like so:

attachment:0e073ace-1f49-4a9a-b16b-79a00c9f9242:InteractiveDevelopment.mp4

However, since we have the `self.embed` command, the animation halts after the square is made and a python shell is opened. All the code after that point is not run, so you can try copying and pasting some of these lines of code into the shell to mess around:

```
self.play(ReplacementTransform(square, circle))
self.wait()
self.play(circle.animate.stretch(4, 0))
self.play(Rotate(circle, 90 * DEG))
self.play(circle.animate.shift(2 * RIGHT).scale(0.25))

text = Text("""
    In general, using the interactive shell
    is very helpful when developing new scenes
""")
self.play(Write(text))
```

In the interactive shell, you can just type play, add, remove, clear, wait, save_state and restore instead of self.play, self.add, self.remove, etc. We can even customize a scene to be be responsive to mouse and keyboard interactions:

```
always(circle.move_to, self.mouse_point)
```

Now we're going to look through the rest of the scene examples to give you some ideas on things you can make with the software. Feel free to edit your versions of the classes to get a feel for writing manim yourself:

Animating Methods

```
class AnimatingMethods(Scene):
def construct(self):
        grid = OldTex(r"\pi").get_grid(10, 10, height=4)
        self.add(grid)

# You can animate the application of mobject methods with the ".animate" syn
tax:self.play(grid.animate.shift(LEFT))

# Alternatively, you can use the older syntax by passing the method and then
the arguments to the scene's "play" function:self.play(grid.shift, LEFT)

# Both of those will interpolate between the mobject's initial state and whatev
er happens when you apply that method.
# For this example, calling grid.shift(LEFT) would shift the grid one unit to the l
eft, but both of the previous calls to "self.play" animate that motion. The same
applies for any method, including those setting colors.self.play(grid.animate.s
et_color(YELLOW))
        self.wait()
        self.play(grid.animate.set_submobject_colors_by_gradient(BLUE, GREE
N))
        self.wait()
```

```
        self.play(grid.animate.set_height(TAU - MED_SMALL_BUFF))
        self.wait()

# The method Mobject.apply_complex_function lets you apply arbitrary compl
ex functions, treating the points defining the mobject as complex numbers.sel
f.play(grid.animate.apply_complex_function(np.exp), run_time=5)
        self.wait()

# Even more generally, you could apply Mobject.apply_function,# which takes
in functions from R^3 to R^3self.play(
            grid.animate.apply_function(
lambda p: [
                    p[0] + 0.5 * math.sin(p[1]),
                    p[1] + 0.5 * math.sin(p[0]),
                    p[2]
                ]
            ),
            run_time=5,
        )
        self.wait()
```

We can run this using `manimgl example_scenes.py AnimatingMethods` :

attachment:2795def5-17a6-4412-ac4a-da7229a64ece:AnimatingMethods.mp4

Text Example

```
class TextExample(Scene):
    def construct(self):
```

```
        # To run this scene properly, you should have the "Consolas" font in your
computer
        # for full usage, you can see https://github.com/3b1b/manim/pull/680
        text = Text("Here is a text", font="Consolas", font_size=90)
        difference = Text(
            """
            The most important difference between Text and TexText is that\n
            you can change the font more easily, but can't use the LaTeX grammar
            """,
            font="Arial", font_size=24,
            # t2c is a dict that you can choose color for different text
            t2c={"Text": BLUE, "TexText": BLUE, "LaTeX": ORANGE}
        )
        VGroup(text, difference).arrange(DOWN, buff=1)
        self.play(Write(text))
        self.play(FadeIn(difference, UP))
        self.wait(3)

        fonts = Text(
            "And you can also set the font according to different words",
            font="Arial",
            t2f={"font": "Consolas", "words": "Consolas"},
            t2c={"font": BLUE, "words": GREEN}
        )
        fonts.set_width(FRAME_WIDTH - 1)
        slant = Text(
            "And the same as slant and weight",
            font="Consolas",
            t2s={"slant": ITALIC},
            t2w={"weight": BOLD},
            t2c={"slant": ORANGE, "weight": RED}
        )
        VGroup(fonts, slant).arrange(DOWN, buff=0.8)
        self.play(FadeOut(text), FadeOut(difference, shift=DOWN))
        self.play(Write(fonts))
        self.wait()
```

```
        self.play(Write(slant))
        self.wait()
```

attachment:a2e1184d-fc03-4d48-b961-1121981b18fd:TextExample.mp4

Tex Transform Example

```
class TexTransformExample(Scene):
    def construct(self):
        # Tex to color map
        t2c = {
            "A": BLUE,
            "B": TEAL,
            "C": GREEN,
        }
        # Configuration to pass along to each Tex mobject
        kw = dict(font_size=72, t2c=t2c)
        lines = VGroup(
            Tex("A^2 + B^2 = C^2", **kw),
            Tex("A^2 = C^2 - B^2", **kw),
            Tex("A^2 = (C + B)(C - B)", **kw),
            Tex(R"A = \sqrt{(C + B)(C - B)}", **kw),
        )
        lines.arrange(DOWN, buff=LARGE_BUFF)

        self.add(lines[0])
        # The animation TransformMatchingStrings will line up parts
        # of the source and target which have matching substring strings.
        # Here, giving it a little path_arc makes each part rotate into
        # their final positions, which feels appropriate for the idea of
```

```
        # rearranging an equation
        self.play(
            TransformMatchingStrings(
                lines[0].copy(), lines[1],
                # matched_keys specifies which substring should
                # line up. If it's not specified, the animation
                # will align the longest matching substrings.
                # In this case, the substring "^2 = C^2" would
                # trip it up
                matched_keys=["A^2", "B^2", "C^2"],
                # When you want a substring from the source
                # to go to a non-equal substring from the target,
                # use the key map.
                key_map={"+": "-"},
                path_arc=90 * DEG,
            ),
        )
        self.wait()
        self.play(TransformMatchingStrings(
            lines[1].copy(), lines[2],
            matched_keys=["A^2"]
        ))
        self.wait()
        self.play(
            TransformMatchingStrings(
                lines[2].copy(), lines[3],
                key_map={"2": R"\sqrt"},
                path_arc=-30 * DEG,
            ),
        )
        self.wait(2)
        self.play(LaggedStartMap(FadeOut, lines, shift=2 * RIGHT))

        # TransformMatchingShapes will try to line up all pieces of a
        # source mobject with those of a target, regardless of the
        # what Mobject type they are.
```

```
        source = Text("the morse code", height=1)
        target = Text("here come dots", height=1)
        saved_source = source.copy()

        self.play(Write(source))
        self.wait()
        kw = dict(run_time=3, path_arc=PI / 2)
        self.play(TransformMatchingShapes(source, target, **kw))
        self.wait()
        self.play(TransformMatchingShapes(target, saved_source, **kw))
        self.wait()
```

attachment:d6c562c3-47fa-46d8-805b-57039c9868a9:TexTransformExample.mp4

Tex Indexing Example

```
class TexIndexing(Scene):
    def construct(self):
        # You can index into Tex mobject (or other StringMobjects) by substrings
        equation = Tex(R"e^{\pi i} = -1", font_size=144)

        self.add(equation)
        self.play(FlashAround(equation["e"]))
        self.wait()
        self.play(Indicate(equation[R"\pi"]))
        self.wait()
        self.play(TransformFromCopy(
            equation[R"e^{\pi i}"].copy().set_opacity(0.5),
            equation["-1"],
```

```
            path_arc=-PI / 2,
            run_time=3
        ))
        self.play(FadeOut(equation))

        # Or regular expressions
        equation = Tex("A^2 + B^2 = C^2", font_size=144)

        self.play(Write(equation))
        for part in equation[re.compile(r"\w\^2")]:
            self.play(FlashAround(part))
        self.wait()
        self.play(FadeOut(equation))

        # Indexing by substrings like this may not work when
        # the order in which Latex draws symbols does not match
        # the order in which they show up in the string.
        # For example, here the infinity is drawn before the sigma
        # so we don't get the desired behavior.
        equation = Tex(R"\sum_{n = 1}^\infty \frac{1}{n^2} = \frac{\pi^2}{6}", font
_size=72)
        self.play(FadeIn(equation))
        self.play(equation[R"\infty"].animate.set_color(RED))  # Doesn't hit the inf
inity
        self.wait()
        self.play(FadeOut(equation))

        # However you can always fix this by explicitly passing in
        # a string you might want to isolate later. Also, using
        # \over instead of \frac helps to avoid the issue for fractions
        equation = Tex(
            R"\sum_{n = 1}^\infty {1 \over n^2} = {\pi^2 \over 6}",
            # Explicitly mark "\infty" as a substring you might want to access
            isolate=[R"\infty"],
            font_size=72
        )
```

```
        self.play(FadeIn(equation))
        self.play(equation[R"\infty"].animate.set_color(RED))  # Got it!
        self.wait()
        self.play(FadeOut(equation))
```

attachment:88c7aae9-e646-46ea-80fe-644a23443234:TexIndexing.mp4

Updaters Example

```
class UpdatersExample(Scene):
    def construct(self):
        square = Square()
        square.set_fill(BLUE_E, 1)

        # On all frames, the constructor Brace(square, UP) will
        # be called, and the mobject brace will set its data to match
        # that of the newly constructed object
        brace = always_redraw(Brace, square, UP)

        label = TexText("Width = 0.00")
        number = label.make_number_changeable("0.00")

        # This ensures that the method deicmal.next_to(square)
        # is called on every frame
        label.always.next_to(brace, UP)
        # You could also write the following equivalent line
        # label.add_updater(lambda m: m.next_to(brace, UP))

        # If the argument itself might change, you can use f_always,
        # for which the arguments following the initial Mobject method
```

```
        # should be functions returning arguments to that method.
        # The following line ensures thst decimal.set_value(square.get_y())
        # is called every frame
        number.f_always.set_value(square.get_width)
        # You could also write the following equivalent line
        # number.add_updater(lambda m: m.set_value(square.get_width()))

        self.add(square, brace, label)

        # Notice that the brace and label track with the square
        self.play(
            square.animate.scale(2),
            rate_func=there_and_back,
            run_time=2,
        )
        self.wait()
        self.play(
            square.animate.set_width(5, stretch=True),
            run_time=3,
        )
        self.wait()
        self.play(
            square.animate.set_width(2),
            run_time=3
        )
        self.wait()

        # In general, you can alway call Mobject.add_updater, and pass in
        # a function that you want to be called on every frame.  The function
        # should take in either one argument, the mobject, or two arguments,
        # the mobject and the amount of time since the last frame.
        now = self.time
        w0 = square.get_width()
        square.add_updater(
            lambda m: m.set_width(w0 * math.sin(self.time - now) + w0)
```

```
        )
        self.wait(4 * PI)
```

attachment:fd234e69-1efd-47fe-85f1-b24f696792aa:UpdatersExample.mp4

Coordinate System Example

```
class CoordinateSystemExample(Scene):
    def construct(self):
        axes = Axes(
            # x-axis ranges from -1 to 10, with a default step size of 1
            x_range=(-1, 10),
            # y-axis ranges from -2 to 2 with a step size of 0.5
            y_range=(-2, 2, 0.5),
            # The axes will be stretched so as to match the specified
            # height and width
            height=6,
            width=10,
            # Axes is made of two NumberLine mobjects.  You can specify
            # their configuration with axis_config
            axis_config=dict(
                stroke_color=GREY_A,
                stroke_width=2,
                numbers_to_exclude=[0],
            ),
            # Alternatively, you can specify configuration for just one
            # of them, like this.
            y_axis_config=dict(
                big_tick_numbers=[-2, 2],
```

```
        )
    )
    # Keyword arguments of add_coordinate_labels can be used to
    # configure the DecimalNumber mobjects which it creates and
    # adds to the axes
    axes.add_coordinate_labels(
        font_size=20,
        num_decimal_places=1,
    )
    self.add(axes)

    # Axes descends from the CoordinateSystem class, meaning
    # you can call call axes.coords_to_point, abbreviated to
    # axes.c2p, to associate a set of coordinates with a point,
    # like so:
    dot = Dot(color=RED)
    dot.move_to(axes.c2p(0, 0))
    self.play(FadeIn(dot, scale=0.5))
    self.play(dot.animate.move_to(axes.c2p(3, 2)))
    self.wait()
    self.play(dot.animate.move_to(axes.c2p(5, 0.5)))
    self.wait()

    # Similarly, you can call axes.point_to_coords, or axes.p2c
    # print(axes.p2c(dot.get_center()))

    # We can draw lines from the axes to better mark the coordinates
    # of a given point.
    # Here, the always_redraw command means that on each new frame
    # the lines will be redrawn
    h_line = always_redraw(lambda: axes.get_h_line(dot.get_left()))
    v_line = always_redraw(lambda: axes.get_v_line(dot.get_bottom()))

    self.play(
        ShowCreation(h_line),
        ShowCreation(v_line),
```

```
        )
        self.play(dot.animate.move_to(axes.c2p(3, -2)))
        self.wait()
        self.play(dot.animate.move_to(axes.c2p(1, 1)))
        self.wait()

        # If we tie the dot to a particular set of coordinates, notice
        # that as we move the axes around it respects the coordinate
        # system defined by them.
        f_always(dot.move_to, lambda: axes.c2p(1, 1))
        self.play(
            axes.animate.scale(0.75).to_corner(UL),
            run_time=2,
        )
        self.wait()
        self.play(FadeOut(VGroup(axes, dot, h_line, v_line)))

        # Other coordinate systems you can play around with include
        # ThreeDAxes, NumberPlane, and ComplexPlane.
```

attachment:dc686317-503a-4948-b5e9-05a4ed8021b8:CoordinateSystemExample.mp4

Graph Example

```
class GraphExample(Scene):
    def construct(self):
        axes = Axes((-3, 10), (-1, 8))
        axes.add_coordinate_labels()
```

```
        self.play(Write(axes, lag_ratio=0.01, run_time=1))

        # Axes.get_graph will return the graph of a function
        sin_graph = axes.get_graph(
            lambda x: 2 * math.sin(x),
            color=BLUE,
        )
        # By default, it draws it so as to somewhat smoothly interpolate
        # between sampled points (x, f(x)).  If the graph is meant to have
        # a corner, though, you can set use_smoothing to False
        relu_graph = axes.get_graph(
            lambda x: max(x, 0),
            use_smoothing=False,
            color=YELLOW,
        )
        # For discontinuous functions, you can specify the point of
        # discontinuity so that it does not try to draw over the gap.
        step_graph = axes.get_graph(
            lambda x: 2.0 if x > 3 else 1.0,
            discontinuities=[3],
            color=GREEN,
        )

        # Axes.get_graph_label takes in either a string or a mobject.
        # If it's a string, it treats it as a LaTeX expression.  By default
        # it places the label next to the graph near the right side, and
        # has it match the color of the graph
        sin_label = axes.get_graph_label(sin_graph, "\\sin(x)")
        relu_label = axes.get_graph_label(relu_graph, Text("ReLU"))
        step_label = axes.get_graph_label(step_graph, Text("Step"), x=4)

        self.play(
            ShowCreation(sin_graph),
            FadeIn(sin_label, RIGHT),
        )
        self.wait(2)
```

```
        self.play(
            ReplacementTransform(sin_graph, relu_graph),
            FadeTransform(sin_label, relu_label),
        )
        self.wait()
        self.play(
            ReplacementTransform(relu_graph, step_graph),
            FadeTransform(relu_label, step_label),
        )
        self.wait()

        parabola = axes.get_graph(lambda x: 0.25 * x**2)
        parabola.set_stroke(BLUE)
        self.play(
            FadeOut(step_graph),
            FadeOut(step_label),
            ShowCreation(parabola)
        )
        self.wait()

        # You can use axes.input_to_graph_point, abbreviated
        # to axes.i2gp, to find a particular point on a graph
        dot = Dot(color=RED)
        dot.move_to(axes.i2gp(2, parabola))
        self.play(FadeIn(dot, scale=0.5))

        # A value tracker lets us animate a parameter, usually
        # with the intent of having other mobjects update based
        # on the parameter
        x_tracker = ValueTracker(2)
        f_always(
            dot.move_to,
            lambda: axes.i2gp(x_tracker.get_value(), parabola)
        )

        self.play(x_tracker.animate.set_value(4), run_time=3)
```

```
        self.play(x_tracker.animate.set_value(-2), run_time=3)
        self.wait()
```

attachment:d499e05f-7d05-4b76-a2cf-dbecac44ea21:GraphExample.mp4

Surface Example

IMPORTANT Before this code is run, in your terminal you need to run `mkdir -p downloads`

```
class SurfaceExample(Scene):
    CONFIG = {
        "camera_class": ThreeDCamera,
    }

    def construct(self):
        surface_text = Text("For 3d scenes, try using surfaces")
        surface_text.fix_in_frame()
        surface_text.to_edge(UP)
        self.add(surface_text)
        self.wait(0.1)

        torus1 = Torus(r1=1, r2=1)
        torus2 = Torus(r1=3, r2=1)
        sphere = Sphere(radius=3, resolution=torus1.resolution)
        # You can texture a surface with up to two images, which will
        # be interpreted as the side towards the light, and away from
        # the light.  These can be either urls, or paths to a local file
        # in whatever you've set as the image directory in
        # the custom_config.yml file
```

```
        # day_texture = "EarthTextureMap"
        # night_texture = "NightEarthTextureMap"
        day_texture = "https://upload.wikimedia.org/wikipedia/commons/thumb/
4/4d/Whole_world_-_land_and_oceans.jpg/1280px-Whole_world_-_land_and_o
ceans.jpg"
        night_texture = "https://upload.wikimedia.org/wikipedia/commons/thum
b/b/ba/The_earth_at_night.jpg/1280px-The_earth_at_night.jpg"

        surfaces = [
            TexturedSurface(surface, day_texture, night_texture)
            for surface in [sphere, torus1, torus2]
        ]

        for mob in surfaces:
            mob.shift(IN)
            mob.mesh = SurfaceMesh(mob)
            mob.mesh.set_stroke(BLUE, 1, opacity=0.5)

        # Set perspective
        frame = self.camera.frame
        frame.set_euler_angles(
            theta=-30 * DEGREES,
            phi=70 * DEGREES,
        )

        surface = surfaces[0]

        self.play(
            FadeIn(surface),
            ShowCreation(surface.mesh, lag_ratio=0.01, run_time=3),
        )
        for mob in surfaces:
            mob.add(mob.mesh)
        surface.save_state()
        self.play(Rotate(surface, PI / 2), run_time=2)
```

```
        for mob in surfaces[1:]:
            mob.rotate(PI / 2)

        self.play(
            Transform(surface, surfaces[1]),
            run_time=3
        )

        self.play(
            Transform(surface, surfaces[2]),
            # Move camera frame during the transition
            frame.animate.increment_phi(-10 * DEGREES),
            frame.animate.increment_theta(-20 * DEGREES),
            run_time=3
        )
        # Add ambient rotation
        frame.add_updater(lambda m, dt: m.increment_theta(-0.1 * dt))

        # Play around with where the light is
        light_text = Text("You can move around the light source")
        light_text.move_to(surface_text)
        light_text.fix_in_frame()

        self.play(FadeTransform(surface_text, light_text))
        light = self.camera.light_source
        self.add(light)
        light.save_state()
        self.play(light.animate.move_to(3 * IN), run_time=5)
        self.play(light.animate.shift(10 * OUT), run_time=5)

        drag_text = Text("Try moving the mouse while pressing d or s")
        drag_text.move_to(light_text)
        drag_text.fix_in_frame()
```

```
        self.play(FadeTransform(light_text, drag_text))
        self.wait()
```

attachment:1fc0c76b-bb98-4186-8434-e32216011d8d:SurfaceExample.mp4

If you want to try dragging around this 3D scene, at the very end add `self.embed()` before you run `manimgl example_scenes.py SurfaceExample`

Opening Manim Example

```
class OpeningManimExample(Scene):
    def construct(self):
        intro_words = Text("""
            The original motivation for manim was to
            better illustrate mathematical functions
            as transformations.
        """)
        intro_words.to_edge(UP)

        self.play(Write(intro_words))
        self.wait(2)

        # Linear transform
        grid = NumberPlane((-10, 10), (-5, 5))
        matrix = [[1, 1], [0, 1]]
        linear_transform_words = VGroup(
            Text("This is what the matrix"),
            IntegerMatrix(matrix),
            Text("looks like")
        )
```

```
        linear_transform_words.arrange(RIGHT)
        linear_transform_words.to_edge(UP)
        linear_transform_words.set_backstroke(width=5)

        self.play(
            ShowCreation(grid),
            FadeTransform(intro_words, linear_transform_words)
        )
        self.wait()
        self.play(grid.animate.apply_matrix(matrix), run_time=3)
        self.wait()

        # Complex map
        c_grid = ComplexPlane()
        moving_c_grid = c_grid.copy()
        moving_c_grid.prepare_for_nonlinear_transform()
        c_grid.set_stroke(BLUE_E, 1)
        c_grid.add_coordinate_labels(font_size=24)
        complex_map_words = TexText("""
            Or thinking of the plane as $\\mathds{C}$,\\\\
            this is the map $z \\rightarrow z^2$
        """)
        complex_map_words.to_corner(UR)
        complex_map_words.set_backstroke(width=5)

        self.play(
            FadeOut(grid),
            Write(c_grid, run_time=3),
            FadeIn(moving_c_grid),
            FadeTransform(linear_transform_words, complex_map_words),
        )
        self.wait()
        self.play(
            moving_c_grid.animate.apply_complex_function(lambda z: z**2),
            run_time=6,
```

```
        )
        self.wait(2)
```

attachment:c84d6f84-c475-4dcc-bbe3-631a9b70ce56:OpeningManimExample.mp4

Command Line Interface Flags

So far once we've made manim code I've been giving you snippets of code to paste into the terminal to run, but now we're going to learn how to use the Command Line Interface ourselves.

To run your Manim files, you need to enter the directory at the same level as `manimlib/` and enter the command in the following format into terminal:

```
manimgl <code>.py <Scene> <flags>
```

- `<code>.py` : The python file you wrote. Needs to be at the same level as `manimlib/`, otherwise you need to use an absolute path or a relative path.
- `<Scene>` : The scene you want to render here. If it is not written or written incorrectly, it will list all for you to choose. And if there is only one `Scene` in the file, this class will be rendered directly.
- `<flags>` : CLI flags.

Here are some useful flags to get you started

- `w` to write the scene to a file.
- `o` to write the scene to a file and open the result.

- `s` to skip to the end and just show the final frame. This is great for making images rather than videos!
  - `so` will save the final frame to an image and show it.
- `n <number>` to skip ahead to the `n`'th animation of a scene.
- `f` to make the playback window fullscreen.

Here are all the supported flags for ManimGL:

flag	abbreviation	function
`--help`	`-h`	Show the help message and exit
`--version`	`-v`	Display the version of manimgl
`--write_file`	`-w`	Render the scene as a movie file
`--skip_animations`	`-s`	Skip to the last frame
`--low_quality`	`-l`	Render at a low quality (for faster rendering)
`--medium_quality`	`-m`	Render at a medium quality
`--hd`		Render at a 1080p quality
`--uhd`		Render at a 4k quality
`--full_screen`	`-f`	Show window in full screen
`--presenter_mode`	`-p`	Scene will stay paused during wait calls until space bar or right arrow

		is hit, like a slide show
`--save_pngs`	`-g`	Save each frame as a png
`--gif`	`-i`	Save the video as gif
`--transparent`	`-t`	Render to a movie file with an alpha channel
`--quiet`	`-q`	
`--write_all`	`-a`	Write all the scenes from a file
`--open`	`-o`	Automatically open the saved file once its done
`--finder`		Show the output file in finder
`--config`		Guide for automatic configuration
`--file_name FILE_NAME`		Name for the movie or image file
`--start_at_animation_number START_AT_ANIMATION_NUMBER`	`-n`	Start rendering not from the first animation, but from another, specified by its index. If you passing two comma separated values, e.g. “3,6”, it will end the rendering at the second value.
`--embed [EMBED]`	`-e`	Creates a new file where the

		line `self.embed` is inserted into the Scenes construct method. If a string is passed in, the line will be inserted below the last line of code including that string.
`--resolution RESOLUTION`	`-r`	Resolution, passed as “WxH”, e.g. “1920×1080”
`--fps FPS`		Frame rate, as an integer
`--color COLOR`	`-c`	Background color
`--leave_progress_bars`		Leave progress bars displayed in terminal
`--video_dir VIDEO_DIR`		Directory to write video
`--config_file CONFIG_FILE`		Path to the custom configuration file
`--log-level LOG_LEVEL`		Level of messages to Display, can be DEBUG / INFO / WARNING / ERROR / CRITICAL

Custom Configuration

We can also create a custom configuration for Manim, which will change the defaults of our programs to be more convenient. Here we’ll make a few changes to our configuration to make the renderings more pretty:

1. Enter your `Manim_Stuff` folder from Finder
2. Go into your `manim` folder
3. Go into your `manimlib` folder
4. Open `default_config.yml`
5. Go to line 51 and change the line to `background_color: "000000"` (this sets the background color to black, rather than gray)
6. Go to line 60 and change the line to `saturation: 1.5` (this makes the colors much more vibrant, very pretty against a black background)
7. Go to line 78 and change the line to `font: "CMU Serif"` (this sets the default font to the iconic Computer Modern from LaTeX)
8. Go to line 79 and change the line to `alignment: "CENTER"` (Animations tend to come out cleaner with center aligned text)
9. Depending on your computer specs, you may want to change the default resolution of renderings. If you're on a video editing workstation like me, you may want to change line 50 to `resolution: (3840, 2160)` to generate 4k animations by default.
10. Press `COMMAND + S` to save and then exit the file.

Now you're all set to make beautiful visuals! To learn how to use Manim in slide decks, check out my Manim Slides tutorial:

How to use Manim Slides