SDES3107

INTRODUCTIONTOPROCESSING

WEEK 4

MONDAY 9-12PM

WHAT IS GENERATIVE DESIGN?

"Generative design is a **design method** in which the output – image, sound, architectural models, animation – is **generated by a set of rules** or an algorithm, normally by **using a computer program**."

Generative design can be used in **architecture** (e.g. to generate building shapes and facades), **graphics** (e.g. using outputs from Processing), **objects** (e.g. to inspire and inform the shapes or appearance of products and things, such as jewellery) **interactive art** (e.g. tracking the movement of people to change the form of a projection), and more...

WHAT IS GENERATIVE DESIGN?

An approach to making art or design using computers

A set of processes we can use to create unique and often unexpected graphics

A method by which we can create endless and unique variations of a design or graphic

Working with Processing to make something creative!

THE OBJECTIVITY ENGINE

James Paterson

For several years James has scanned in over 4000 drawings and imported them into Processing, creating a program that selects from this huge library of sketches and makes random compositions.

This is a great example of someone using **generative design processes - algorithms -** to obtain often **unexpected results**.

(Processing handbook - p.165)

www.presstube.com

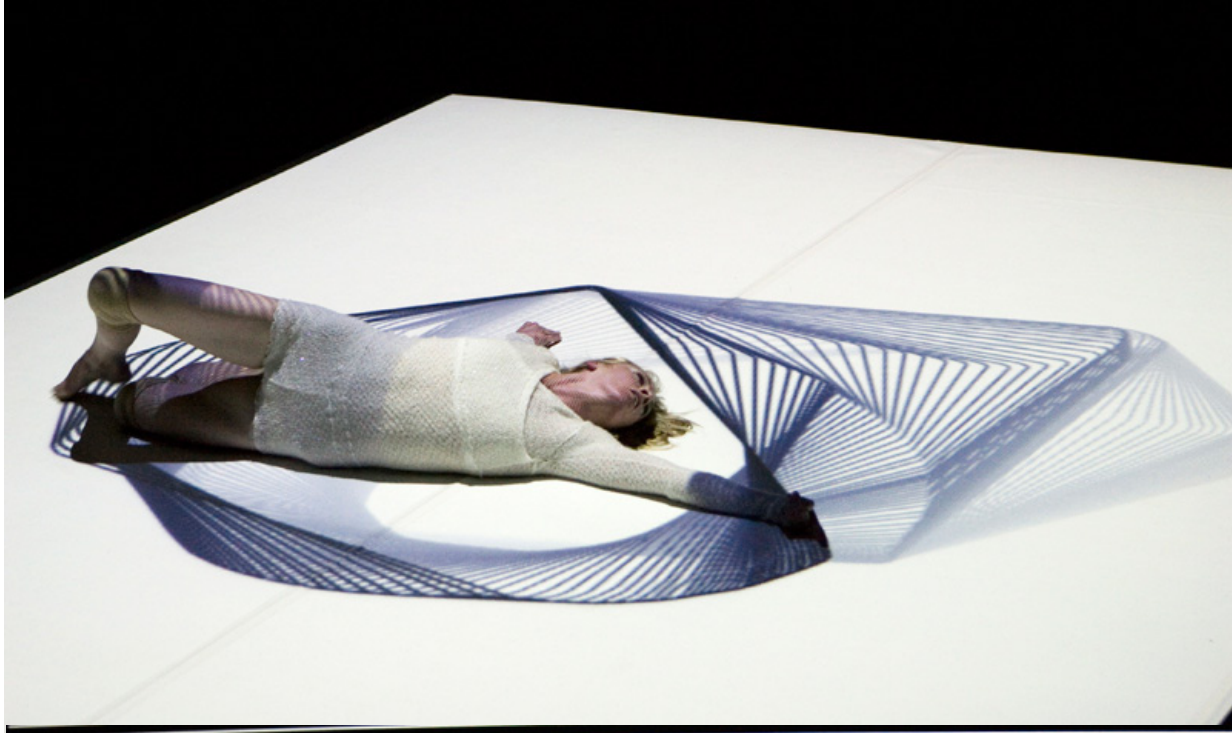
GLOW

Chunky Move

Chunky Move is an Australian company whose work often blends contemporary dance and interactive art. Using camera tracking and motion graphics, they create visually striking performances.

Glow is a good example of **interactive art using generative design processes**. As the dancer moves, the projection changes, **generating new forms** each time a movement is made.

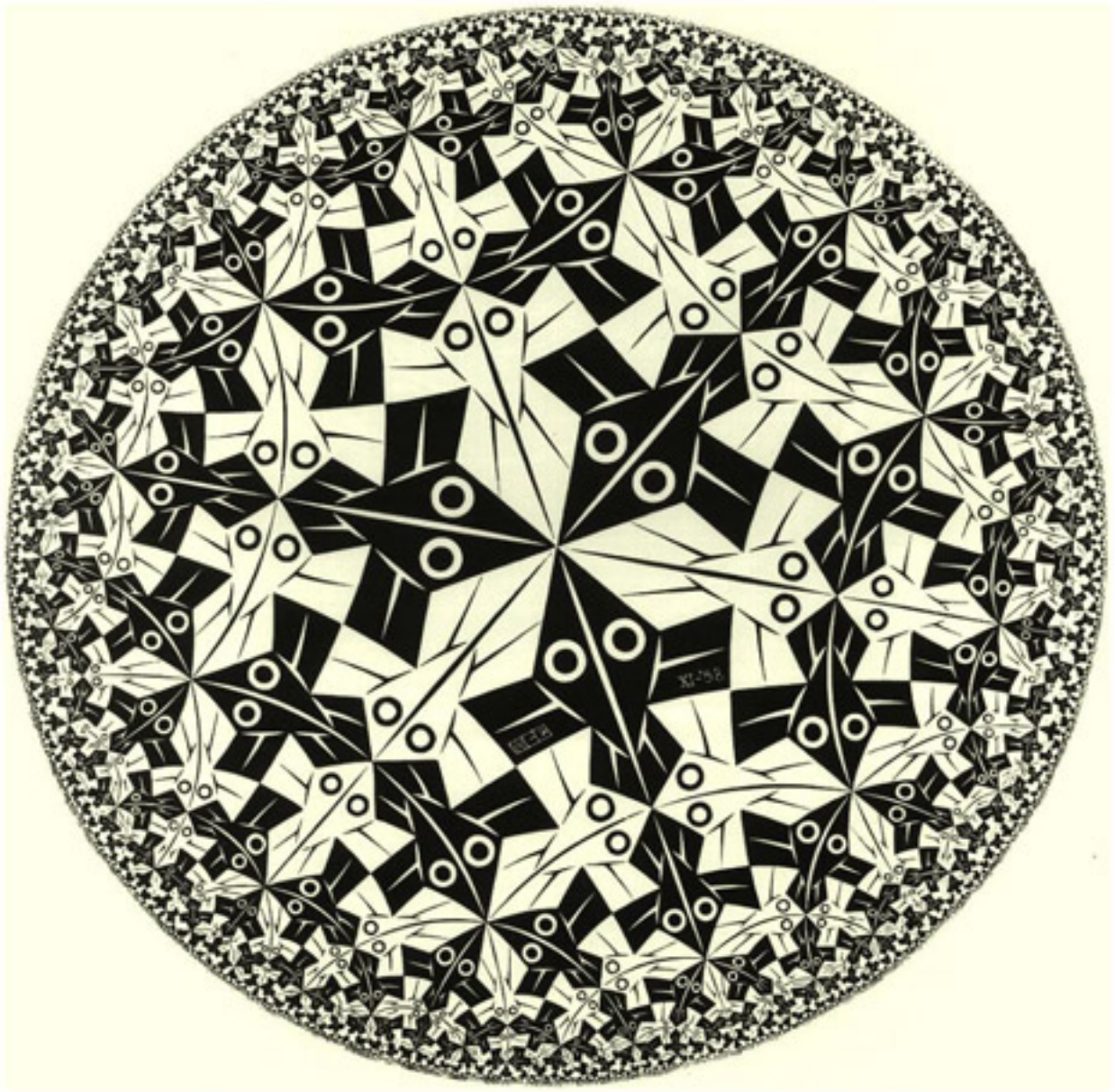
chunkymove.com.au/Our-Works/Current-Productions/Glow.aspx



M.C. ESCHER

Escher engaged in a process we commonly associate with generative design: **recursion**. Recursion is where an image, for example, references itself. Recursion results in **the generation of an infinite series of the same image**.

Escher is different from Nervous System Studio, for example, because he did not directly identify with what we know as generative design today. He never used a computer to make his work. However, **his practice does incorporate a generative design process!**



http://www.pxleyes.com/blog/2010/06/recursion-the-art-and-ideas-behind-m-c-eschers-drawings/

STRUCTURING (p.173-176)

So far we have been making programs that only run once. These are good for static compositions. For dynamic (interactive/animation-based) compositions, we need to use a slightly different structure:

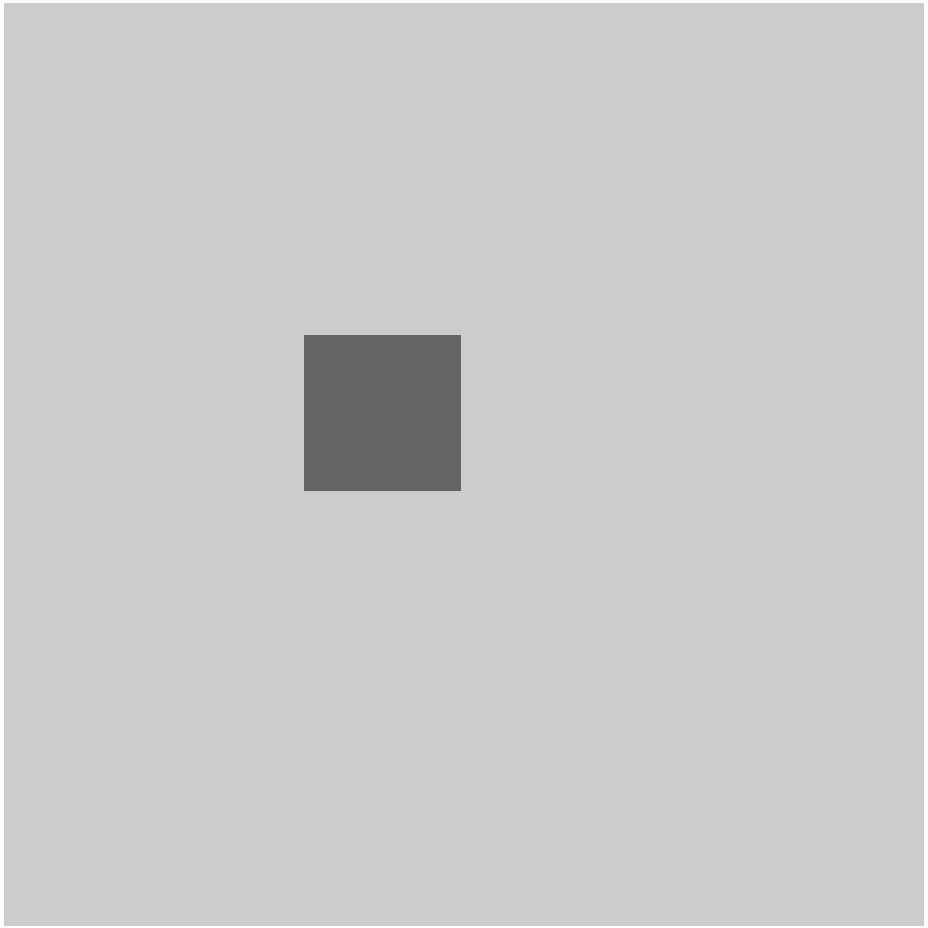
```
size(300,300);
int randomPos =
int(random(300.0));
fill(100);
noStroke();
rect(randomPos,randomPos+
10,50,50);
```

↑
only runs once

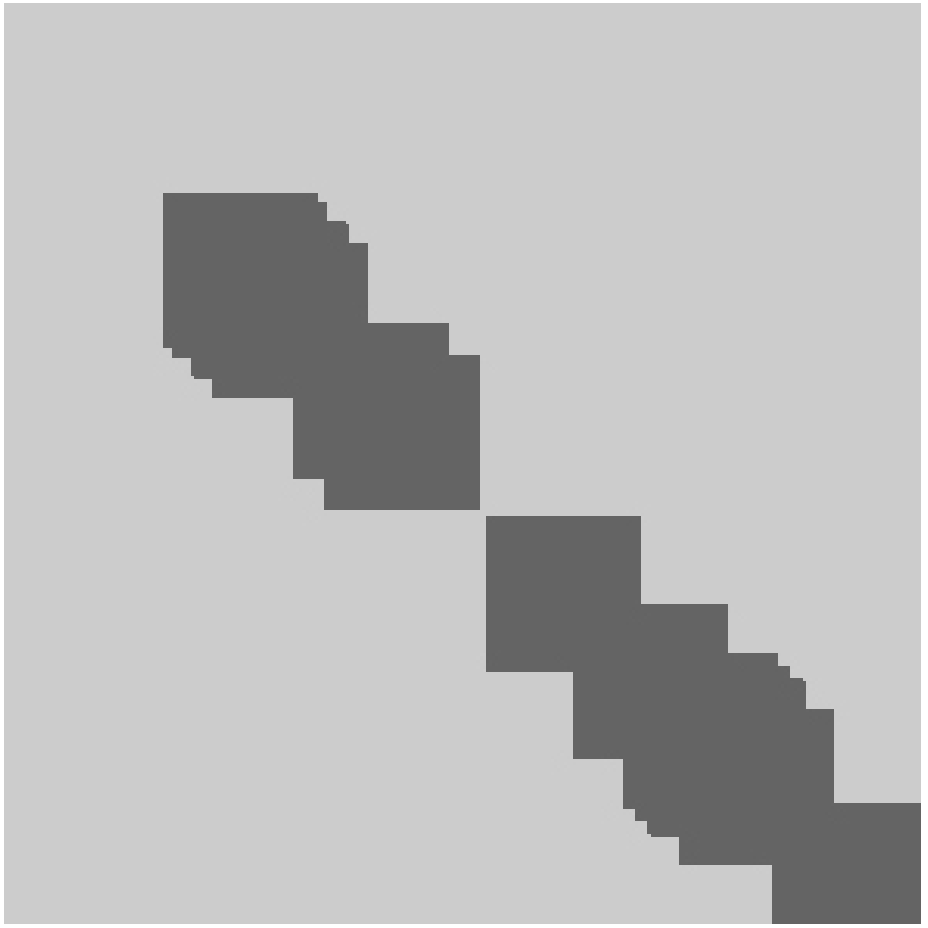
```
void setup() {
size(300,300);
fill(100);
noStroke();
}
void draw() {
int randomPos =
int(random(300.0));
rect(randomPos,randomPos+
10,50,50);
}
```

↑
runs over and over

program that
only runs once



program that runs
over and over



STRUCTURING (p.173-176)

```
int winSize = 300;
```

Most variables should be declared at the top (so they can be accessed anywhere).

```
void setup() {
size(winSize,winSize);
fill(100);
noStroke();
}
```

Think about your code. Anything that **only needs to be set once** should go in **void setup()**.

```
void draw() {
int randomPos =
int(random(300.0));
rect(randomPos,randomPos
+10,50,50);
}
```

Anything that we want to be **drawn over and over**, or **changed in any way as the program runs**, should go in **void draw()**.

STRUCTURING (p.173-176)

There are a few things you can do to control how this works:

Putting noLoop() in the void draw() section stops the program from running over and over.

To stop shapes from building up, put background(255) (or whatever your background is) to “refresh” the background. This gives the illusion of motion!

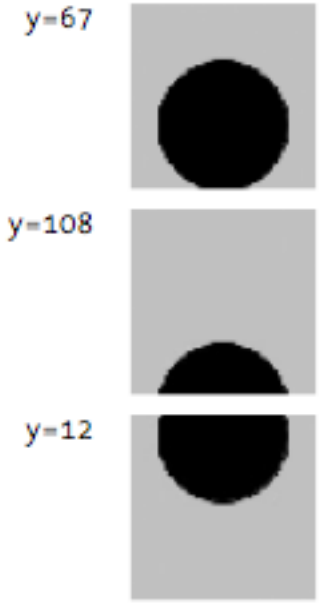
You can slow down the speed of the program by using frameRate().
e.g. frameRate(10).

```
int winSize = 300;

void setup() {
size(winSize, winSize);
fill(100);
noStroke();
frameRate(10);
}

void draw() {
int randomPos =
int(random(300.0));
background(255);
rect(randomPos,randomPos
+10,50,50);
}
```

CONTROLLING MOVEMENT (p.177)

We can control movement of shapes by updating variables in small to large increments.

This is one way of controlling the flow of the draw function.




```
float y = 0.0;

void setup() {
  size(100, 100);
  smooth();
  fill(0);
}

void draw() {
  background(204);
  ellipse(50, y, 70, 70);
  y += 0.5;
  if (y > 150) {
    y = -50.0;
  }
}
```

EXERCISE

20.2. Move a shape from left to right across the screen. When it moves off the right edge, return it to the left.

(p.180)

MOUSE INPUT (p.205-213)

We can add interactivity to our programs by using the mouse input variables: **mouseX** and **mouseY**. They store the **horizontal** and **vertical positions** of the **current mouse position**.

```
void setup() {
size(100,100);
smooth();
noStroke();
}

void draw() {
background(126);
ellipse(mouseX,mouseY,
33,33);
}
```

"refreshes" the screen →

uses the position of the mouse to set the position of an ellipse →

EXERCISE

23.1. Control the position of a shape with the mouse. Strive to create a more interesting relation than one directly mimicking the position of the cursor.

(p.215)

MOUSE INPUT (p.212-213)

We can also make things happen when a **mouse button is pressed**.

To do this we can use an **if structure to test if the mouse has been pressed**:

```
if (mousePressed == true) {
    //something happens here
} else {
    //something else happens
}
```

Example:

```
void setup() {
size(100,100);
smooth();
noStroke();
}

void draw() {
background(204);
if (mousePressed == true) {
    fill(255);
} else {
    fill(0);
}
rect(25,25,50,50);
}
```

DRAWING (p.218-221)

Using the mouse inputs **mouseX**, **mouseY**, and **mousePressed**, we can create our own drawing programs.

We can draw with points, shapes, images, etc...

```
void setup() {
size(100,100);
}

void draw() {
point(mouseX, mouseY);
}
```

Draw only when the mouse is pressed:

```
void setup() {
size(100,100);
stroke(255);
}

void draw() {
if(mousePressed == true){
      line(mouseX, mouseY,
      pmouseX, pmouseY);
      }
}
```

EXERCISE

24.3. Load an image and use it as a drawing tool.

(this is described on p.221)

KEYBOARD INPUT (p.224-227)

Processing can tell when a key has been pressed and also what key it is.

To do this we can use a condition:

if(keyPressed == true){

}

keyPressed returns a boolean data type and is true or false

```
void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}

void draw() {
  background(204);
  if (keyPressed == true) {
    line(20, 20, 80, 80);
  } else {
    rect(40, 40, 20, 20);
  }
}
```

KEYBOARD INPUT (p.224-227)

To tell what key is being pressed we also use a condition:

```
if((keyPressed == true) && (key == 'A')){


}
```

Some keys are 'coded' and need to be referred to in a different way:

```
if(key == CODED){
    if(keyCode == UP){

    }
}
```

Examples of keyCodes:

UP
DOWN
ALT
SHIFT

more can be found at the processing reference page

EXERCISE

25.1. Use the number keys on the keyboard to modify the movement of a line.

EVENTS (p.229-234)

Events are blocks of code that are run when a certain action is happening. Bellow are a list of the events that can be triggered through interaction.

`mousePressed()`	*Code inside this block is run one time when a mouse button is pressed*
`mouseReleased()`	*Code inside this block is run one time when a mouse button is released*
`mouseMoved()`	*Code inside this block is run one time when the mouse is moved*
`mouseDragged()`	*Code inside this block is run one time when the mouse is moved while a mouse button is pressed*
`keyPressed()`	*Code inside this block is run one time when any key is pressed*
`keyReleased()`	*Code inside this block is run one time when any key is released*

EVENTS (p.229-234)

You need to create a function outside of draw and setup for the event:

```
void setup(){

}

void draw(){

}

void mouseDragged(){

}
```

When the event occurs (for example when the mouse is dragged) the code inside the function will be executed.

EXERCISE

26.1. Animate a shape to react when the mouse is pressed and when it is released.

CREATING FUNCTIONS (p.183-193)

We've started to see a new way to view what a 'function' looks like in processing.

```
void draw(){

}
```

We can also create our own functions.

```
void myFunction(  ){

}
```

```
void myFunction(int  x, int y){
    ellipse(x, y, 50, 50);
    line(x, y, x+100, y+50);
}
```

We place this function outside of all other functions like we did with events.

To use this function we need to then write:

```
myFunction(20,30);
```

EXERCISE

21.1. Write a function to draw a shape to the screen multiple times, each at a different position.

PARAMETERIZED SHAPES (p.183-193)

c=15.0

c=25.0

c=35.0

c=45.0

c=55.0

```
float c = 25.0;

void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  arch(c);
}

void arch(float curvature) {
  float y = 90.0;
  strokeWeight(6);
  noFill();
  beginShape();
  vertex(15.0, y);
  bezierVertex(15.0, y-curvature, 30.0, 55.0, 50.0, 55.0);
  bezierVertex(70.0, 55.0, 85.0, y-curvature, 85.0, y);
  endShape();
}
```

We can create shapes or compositions that can be varied simply by changing the parameters we send to them.

EXERCISE

22.1. Write your own function to draw a parameterized arch.