2D Graphics Primer

Sundaram Ramaswamy

April, 2017

Good 'ol drawing

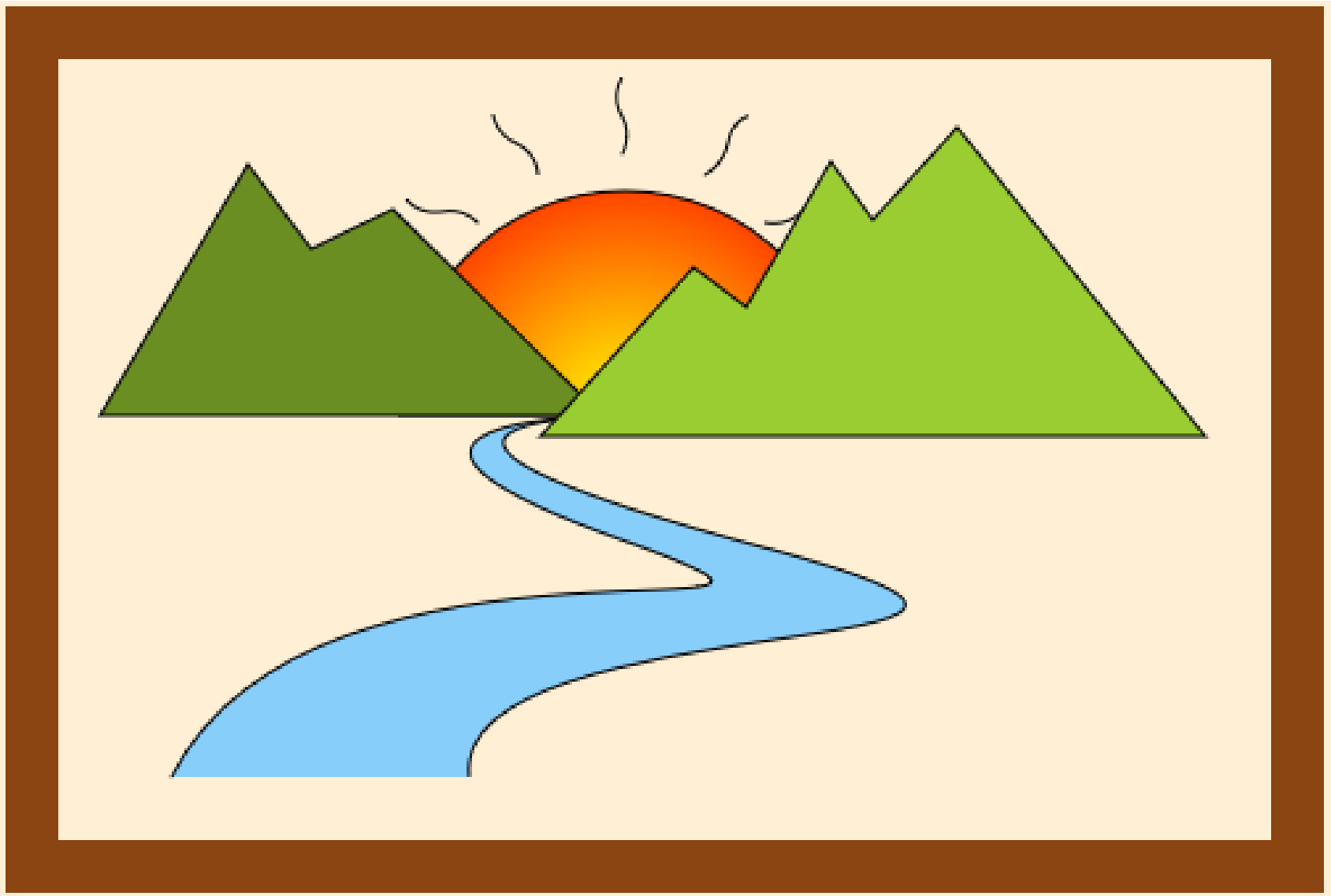
1. Let's draw something simple!
2. Map backwards by learning the theory :)
3. See code to seal the understanding
4. *Rise, repeat*

***Disclaimer**: This is a primer. Chances are you might know a lot more than the presenter and the presented!*

Approach

- Prefer breadth-wise coverage
  - Allows to cover more concepts
  - Pretty pictures are better motivators :)
  - Learning concepts in isolation doesn't encourage you to jump-in
  - Beginners wouldn't appricate digging deep; can always do it alone once motivated
- Favour intuition over rigour
  - Tries to be an "enabler" not a *Reference Guide*
- Concepts are API-agnostic
  - D2D used here but works just as well in say HTML5 canvas, SDL, etc.

Let's Draw This



Dissection 1

Git train: *Stroke* station, coming up!

```
git checkout tags/Stroke
```

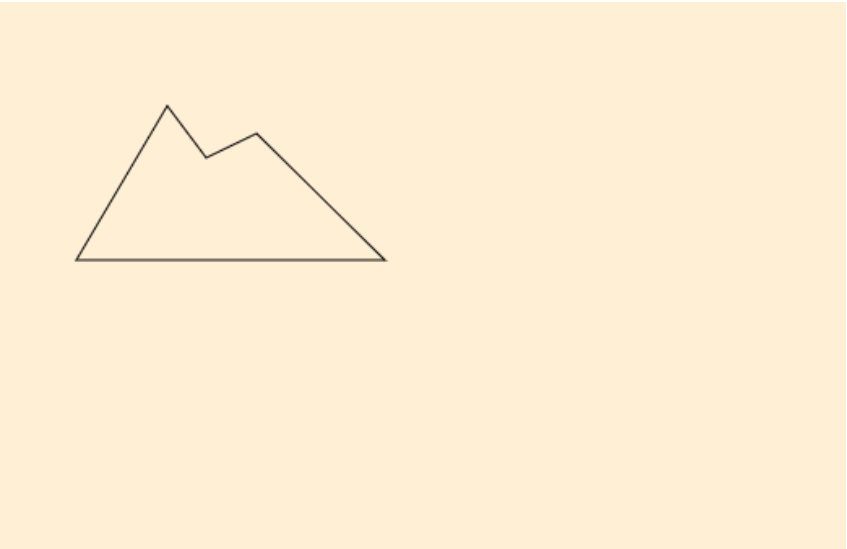


Figure 2: Paths and strokes

Concept: Path / Geometry

- A *mathy* way to remember an artist's work
  - Store every point
    - Where did he put his brush down?
  - Store every hand move
    - Did he draw a line or a curve?

  Example: move to `(5, 5)`, line to `(10, 5)`

- It's just data about **shapes**
  - Hear path/geometry, think **shapes**
  - Data by itself
    - Doesn't talk about how it's used
    - Doesn't depend on hardware/device
    - is plain-old data; no magic
- Once stored this way, any *reader* can imitate the artist
  - Reader may be a human, computer, graph plotter, ...

Path (contd.)

- Commands: `move to`, `line to`, `curve to`, `arc to`, ...
  - Curves are quadratic and cubic Bézier curves
- Each figure has `begin` and `end` (with/out `close`)
  - `close` closes by drawing a line between first and last points
- Don't be intimidated by path drawing languages
  - Just shorthand notations for these commands
  - It's cumbersome to construct figures otherwise
  - XAML: `M 10, 100 L 20, 100`
  - PDF: `10 100 m 20 100 l`
- Represent any artibrary shape: polygon
- Multiple disjoint figures within a single path geometry

Drawing paths: stroke

One, obvious, way to use paths: draw them

Stroke: trace the outline of the geometry's figures.

The brush used by the artist has

- Colour
- Thickness
- Dash pattern: think *dotted line*
- Line cap style
  - Butt cap `|---|`
  - Round cap `(---)`
  - Square cap `|  ---  |`
- Line join style
  - Round join `u`
  - Round join `\_/`
  - Miter join `V`
- Miter limit: max height when joining two line ends

Dissection 2

Git train: next stop, *Fill* station!

```
git checkout tags/Fill
```

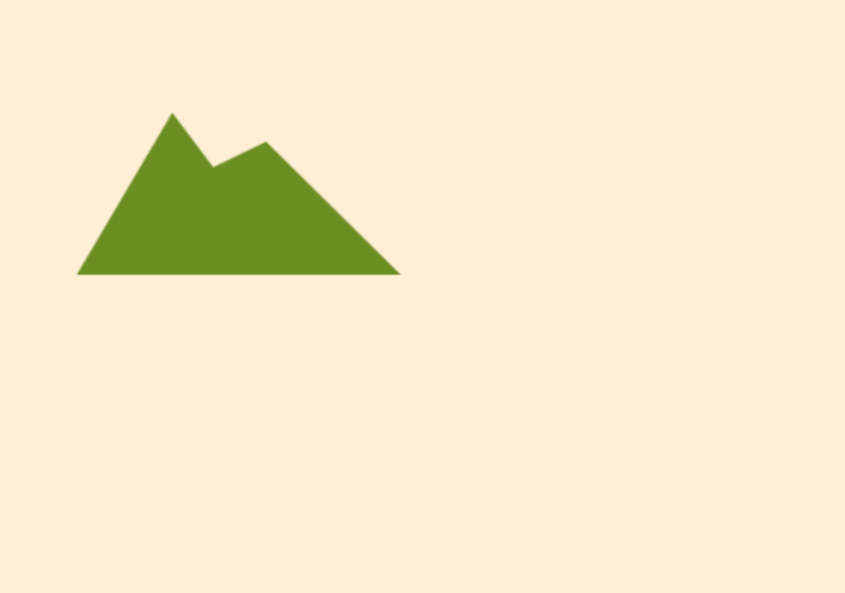


Figure 3: Fills

Drawing paths: fill

- Filling the interior of a path with some brush
  - When unclosed, it's implicitly closed by D2D
  - PDF doesn't honour fill, if unclosed
- But wait, what's *inside* for
  - One sub-path inside another?
  - Self-intersecting paths?
- Winding rule
  - Even-odd (cheaper)
  - Non-zero

Even = fill, Odd = no fill

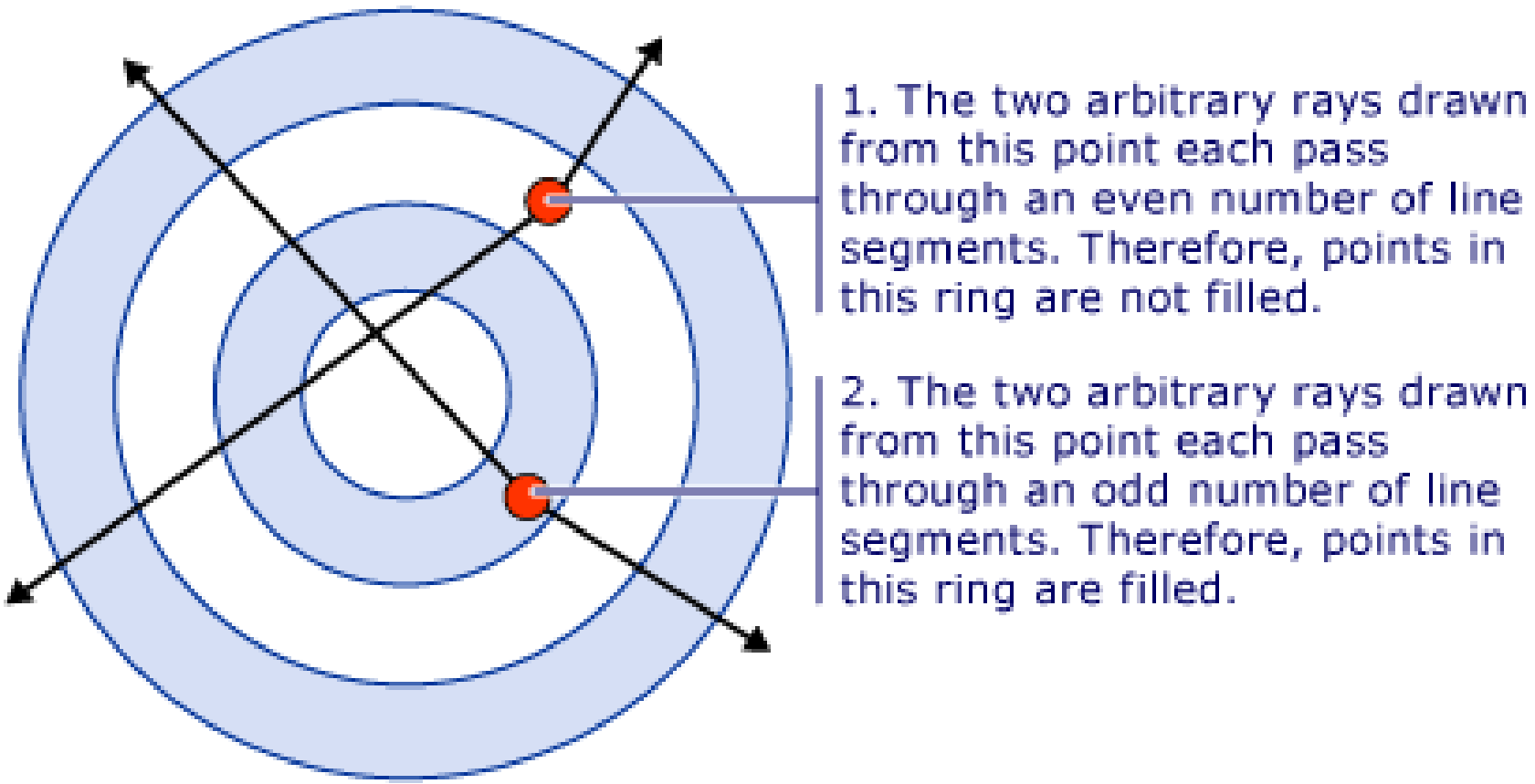


Figure 4: Even Odd rule

Credit: *MSDN*

Non-zero Fill

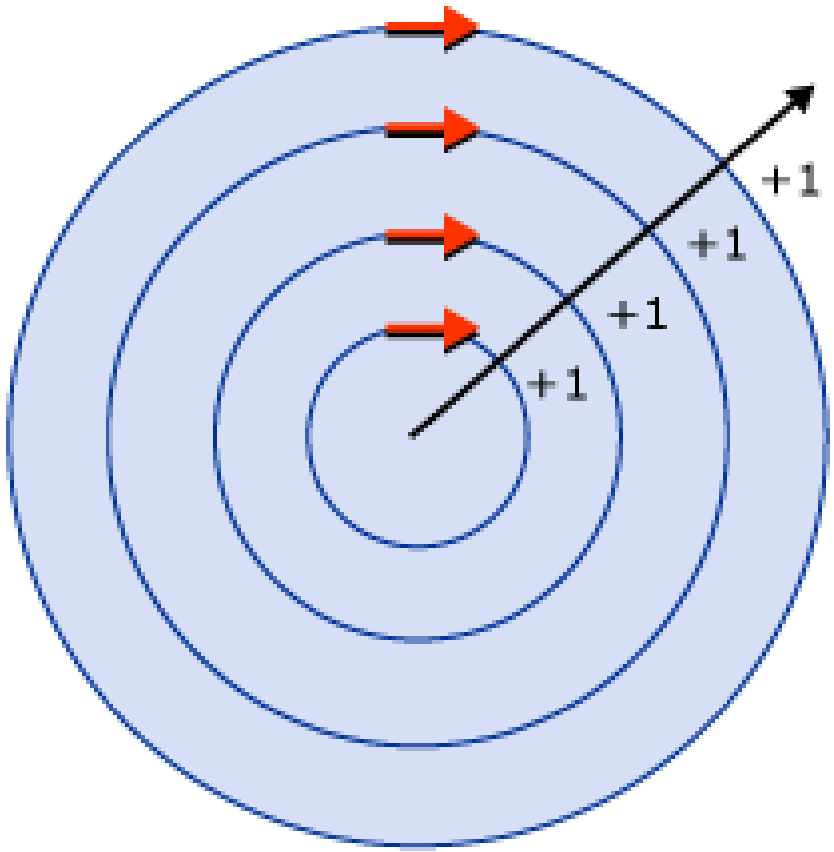


Figure 5: Non-zero rule

Credit: *MSDN*

Resource: Brushes

Figures are fine but what about the colours used to draw?

***Brushes**, also called, **pens** do stroking & filling.*

Didn't speak about its colour thus far; it can be

- Constant
  - Solid: refers the same colour always
- Varies by position
  - Gradient: get colour based on a function of position
    - Linear
    - Radial
  - Bitmap
    - Get colour from a lookup table

Brush Types

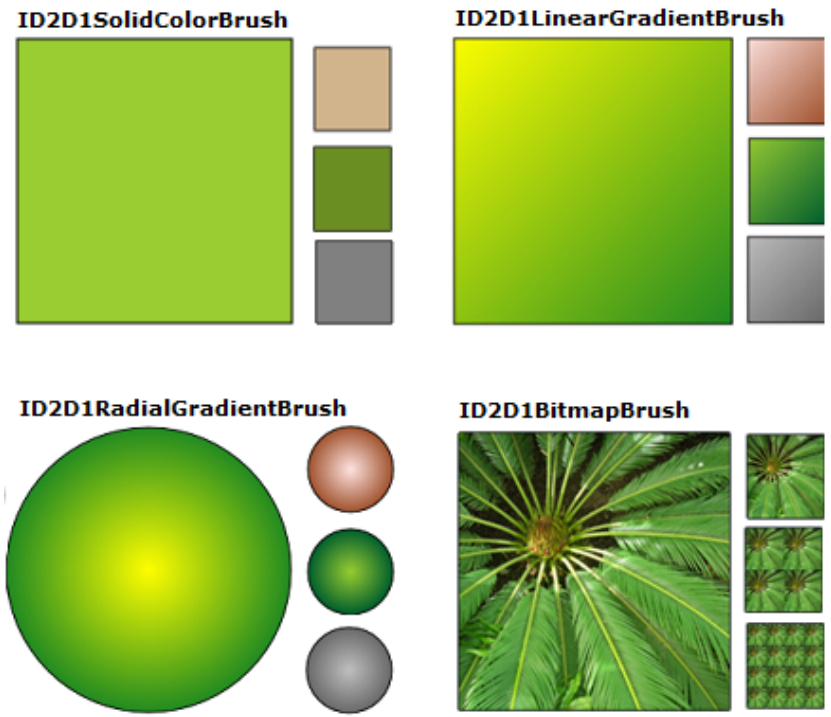


Figure 6: Brush Types

Credit: *MSDN*

Concept: Colours

- Need a precise way of representing colours
- Colour spaces
  - RGB
  - HSV and HSL
  - CMYK
  - many more
- Most rendering APIs work in RGB
  - Additive colour system
  - This is how display systems work
  - Easy math: simple calculations
  - Unintuitive interpolation
- HSV is preferred by artists
  - Most aesthetically pleasing
  - Better for colour pickers
  - Intuitive interpolation

Play Time

git clone
https://bitbucket.org/rmsundaram/tryouts.git

Play with

- `CG\WebGL\CrystalBall\crystal_ball.html`
- `CG\Misc\hsv_wheel.html`

Pixel formats

- Colour value is math; its realisation in hardware leads to

  Pixel: *picture element*

- Irrespective of colour space, data type decides richness
  - Wider the format, richer the gamut
- Integer formats: 0 darkest; `max(channel width)` brightest
  - 32-bit pixel formats: `A8 R8 G8 B8`, `R8 G8 B8 A8`, ...
  - 24-bit pixel formats: `R8 G8 B8`, ...
  - 16-bit pixel formats: `R5 G6 B5`, `A4 R4 G4 B4`, `R5 G5 B5`, ...
  - 8-bit pixel formats: indexing a predefined colour palette
- Floating-point formats: `0.0` darkest, `1.0` brightest
  - Costly: `4 * 32 = 128` bits for RGBA
  - Better for image manipulation
- If exceeds during calculation, it's `clamp` ed by `min` and `max`

Dissection 3

Git train: *Gradient* station, coming up!

```
git checkout tags/Gradient
```

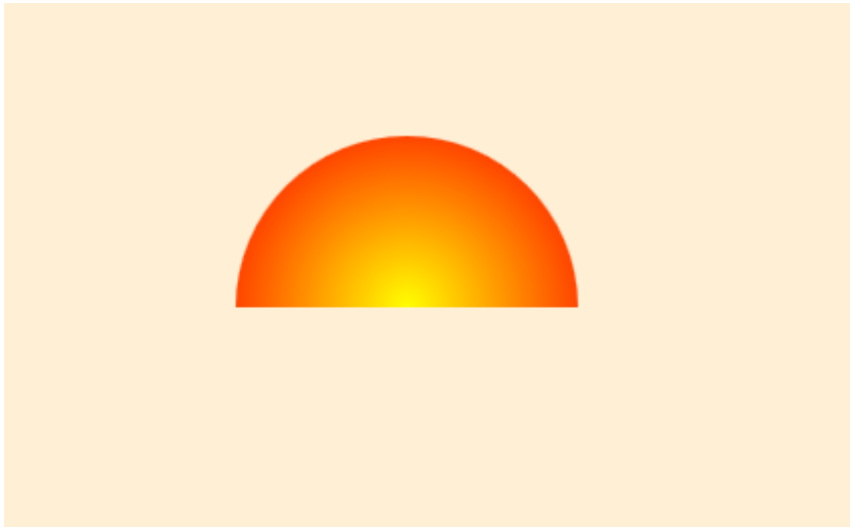
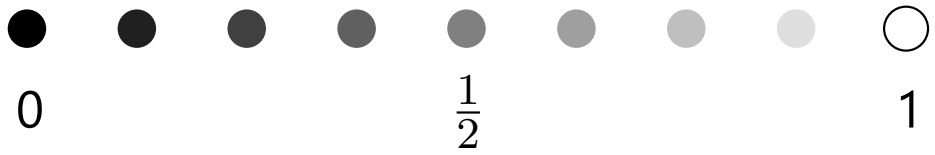


Figure 7: Gradient colours

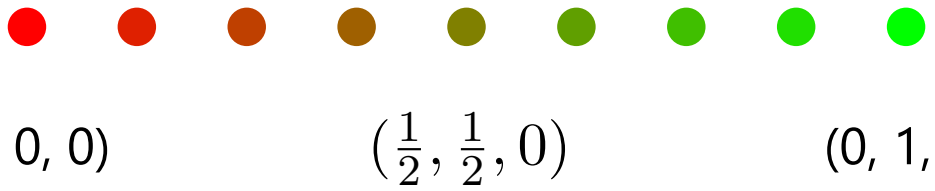
Concept: Interpolation



- Colour at 0 is black `(0, 0, 0)`
- Colour at 1 is white `(1, 1, 1)`
- What's the colour at ½?
- It's pure grey: `(½, ½, ½)`. But how?

Interpolation: *Given two values, guess/find values in between*

Works with any values: position (any dimension), temperature, say even chilli hotness (scoville heat index), etc.



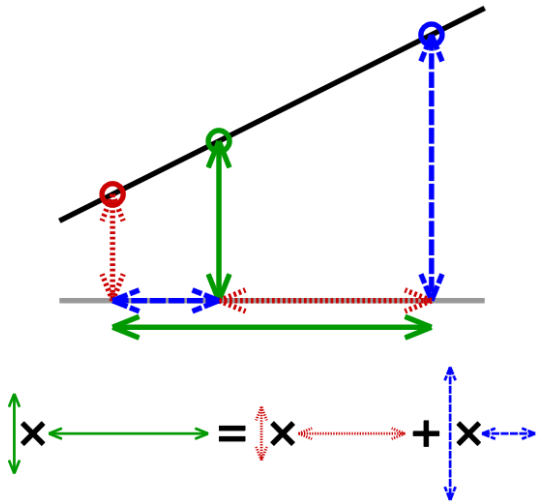
Linear Interpolation a.k.a lerp

$$\boxed{V(t) = (1-t)V_0 + tV_1}$$

```
t              0              1
  |--------P--x------Q------------->
X 0            9   ?          19
```

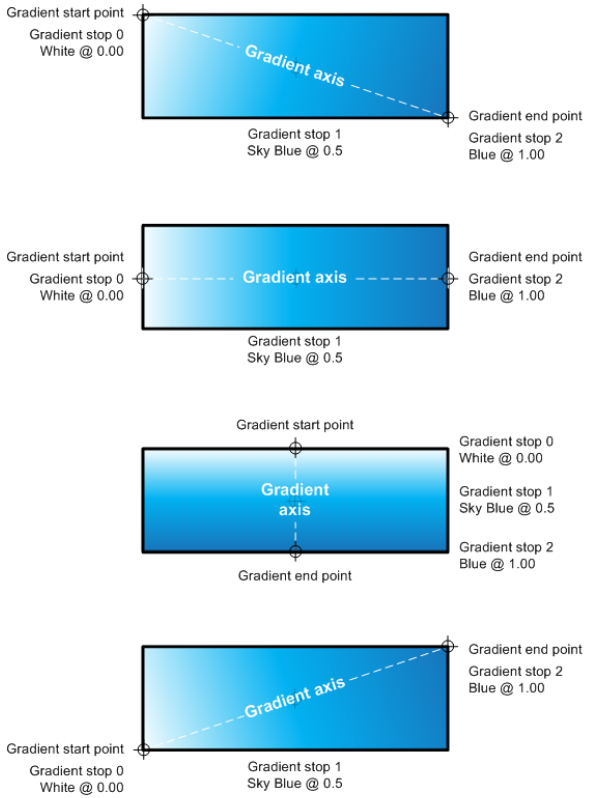
Given P and Q, find the value at $t = 0.3$

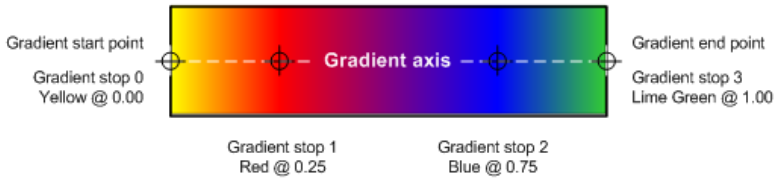
$$V(0.3) = 0.7P + 0.3Q = 0.7(9) + 0.3(19) = 12$$



Linear Gradient Brushes

Interpolates colours (stops) along an axis (line)





Credit: *MSDN*

Radial Gradient Brushes

Think of it as linear interpolation between concentric circles

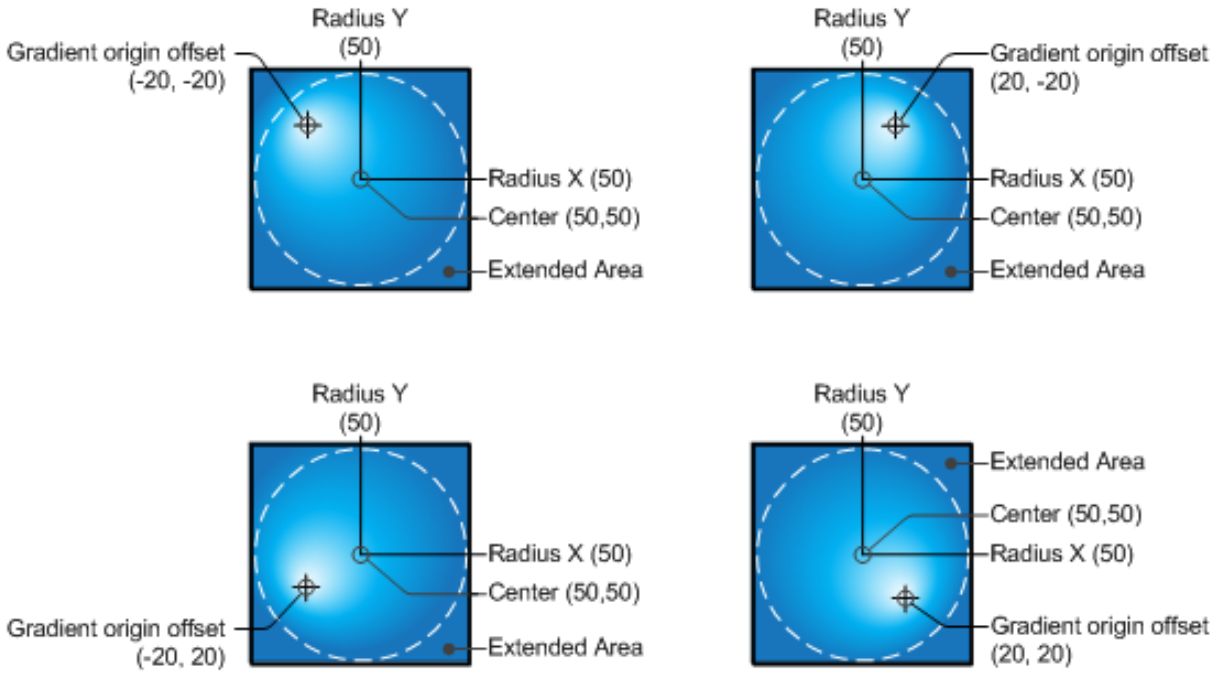


Figure 9: Linear gradient

Credit: *MSDN*

Bitmap Brushes

- Works by indexing the lookup table — image, buffer, texture, surface
- Extend modes a.k.a **how to tile?**
  - Clamp
  - Wrap
  - Mirror
- Interpolation modes a.k.a **how to scale?**
  - Linear
  - Nearest-neighbour

Brush Properties

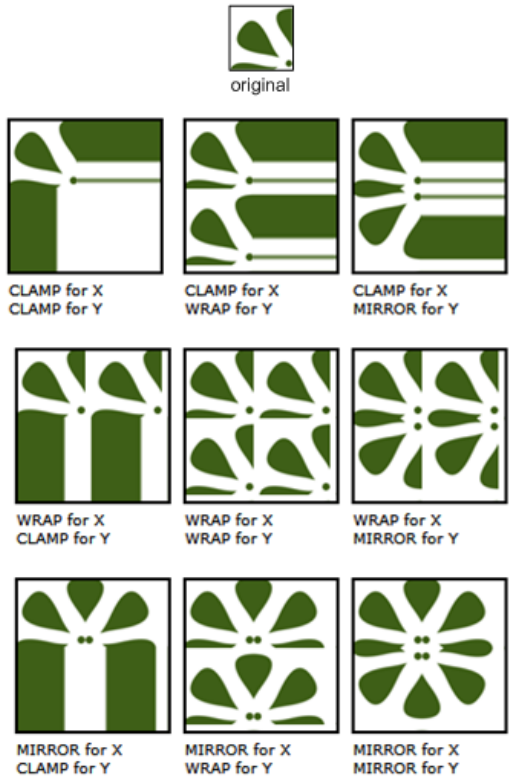


Figure 10: Brush parameters

Shading Patterns

- *Linear* and *radial* are really subclasses of a general class

  Computing an image on-the-fly with math equations and parameters

- Make a bitmap brush out of it
- Can be very *mathy* and need to understand the pattern involved

  `git clone`
  `https://bitbucket.org/rmsundaram/tryouts.git`

Play with `CG\WebGL\coons_patch.html`

Transformations

- Concise representation of changes to points
- Usual operations
  - Scale
  - Rotate
  - Translate
  - Shear
  - Flip/Mirror
- Matrices are used extensively in all graphics APIs
  - Can concatenate multiple transforms into one complex transform
- Internalize them by playing with simple “Hello, World” program
- 2D Transforms 101: a supplementary presentation with interactive animations

Dissection 4

Git train: final stop, *Clip* station!

```
git checkout tags/Clip
```

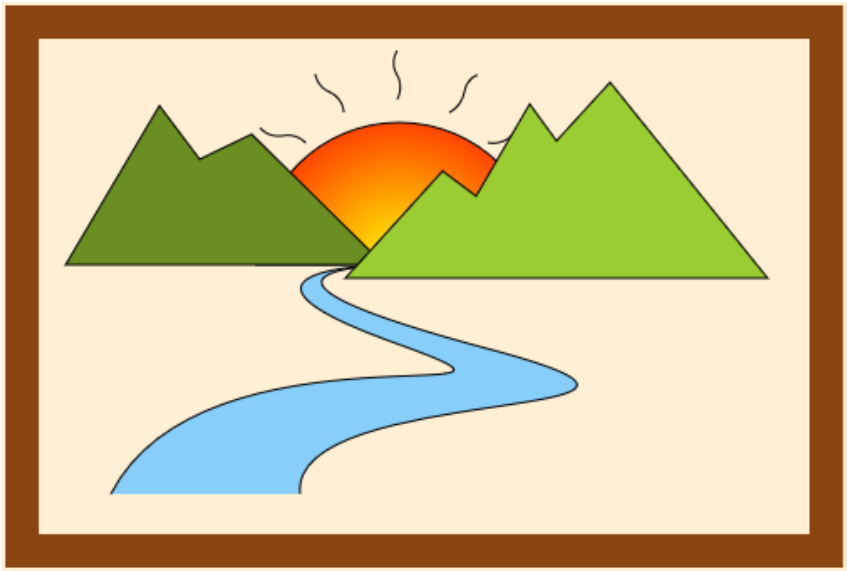


Figure 11: Path-based Clip

Clipping

- Drawing — stroking & filling — by default is conceptually boundless
- In reality, you're bound by the paper / canvas
- On a computer, you're bound by the surface dimension

***Clip**: additional bounds to drawing operations*

- Can be *any* geometry (think *shape*), not necessarily rectangles
- Rectangles are usually faster though
- *Inside*ness is decided by the same rules as *fill*
- Clips are combined by intersection