# **Game Technology**

Lecture 10 – 19.12.2014 Procedural Content Generation





Dr.-Ing. Florian Mehm Dipl-Inf. Robert Konrad

Prof. Dr.-Ing. Ralf Steinmetz KOM - Multimedia Communications Lab

# **Preliminary timetable**



Lecture No.	Date	Topic
1	17.10.2014	Basic Input & Output
2	24.10.2014	Timing & Basic Game Mechanics
3	31.10.2014	Software Rendering 1
4	07.11.2014	Software Rendering 2
5	14.11.2014	Basic Hardware Rendering
6	21.11.2014	Animations
7	28.11.2014	Physically-based Rendering
8	05.12.2014	Physics 1
9	12.12.2014	Physics 2
10	19.12.2014	Procedural Content Generation
11	16.01.2015	Compression & Streaming
12	23.01.2015	Multiplayer
13	30.01.2015	Audio
14	06.02.2015	Scripting
15	13.02.2015	AI

# **Example: No Man's Sky**





# **History**



#### **Elite**

- **1984**
- 8 galaxies with 256 planets each
- Generated galaxies, planets including names and properties
- BBC Micro: max. 128 KB Memory, Elite was 52 KB of disk space

#### **Minecraft**

- Official Release 2011
- Generates terrain including placement of settlements, resources, ... Procedurally
- Sold for 2.5 billion USD to Microsoft in 2014





# **History**

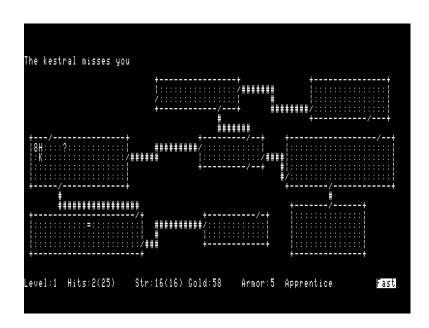


# Rogue & Roguelikes

- Original released in 1980
- Generated a new dungeon each time the player went to a new level

# Algorithm for constructing the levels

 http://kuoi.asui.uidaho.edu/~kamikaz e/GameDesign/art07\_rogue\_dungeo n.php



# **History - Rogue**



- 1. Divide the map into a grid (Rogue uses 3x3, but any size will work).
- 2. Give each grid a flag indicating if it's "connected" or not, and an array of which grid numbers it's connected to.
- 3. Pick a random room to start with, and mark it "connected".
- 4. While there are unconnected neighbor rooms, connect to one of them, make that the current room, mark it "connected", and repeat.
- 5. While there are unconnected rooms, try to connect them to a random connected neighbor (if a room has no connected neighbors yet, just keep cycling, you'll fill out to it eventually).
- 6. All rooms are now connected at least once.
- 7. Make 0 or more random connections to taste; I find rnd(grid\_width) random connections looks good.
- 8. Draw the rooms onto the map, and draw a corridor from the center of each room to the center of each connected room, changing wall blocks into corridors. If your rooms fill most or all of the space of the grid, your corridors will very short just holes in the wall.
- 9. Scan the map for corridor squares with 2 bordering walls, 1-2 bordering rooms, and 0-1 bordering corridor, and change those to doors.
- 10. Place your stairs up in the first room you chose, and your stairs down in the last room chosen in step 5. This will almost always be a LONG way away.

# **Principles**



#### **Structure**

- Many PCG algorithms can create instances of classes of objects
- One type of house, tree, clothing, ...
- Recognizable structure in each instance
- Structured way of deriving an instance

#### Randomness

- Not a defining characteristic of PCG
- But often a central component
- Has the advantages of fooling the eye, adding replayability



# **Classification of Generators**



#### Online versus offline

- Online: Every time a new result
- Offline: Can be evaluated by an artist/designer

# **Necessary versus optional**

- Necessary: The game would not work without PCG
- Optional: Background objects, ambient sounds, ...

# Degree and dimensions of control

- How much control is needed?
- For whom? Game designer, artist, end user?

### **Classification of Genres**



### **Generic versus adaptive**

- Generic: One size fits all
- Adaptive: React to players, build better challenges, adjust for play preferences

#### Stochastic versus deterministic

- Stochastic: Variation, Replayability
- Deterministic: Can be checked, repeated, the same for each player
  - "Arse" galaxy in Elite

#### **Constructive versus Generate-and-test**

- Constructive: Carry out PCG once
- Generate-and-test: Carry out PCG repeatedly

# Automatic generation versus mixed authorship

• Mixed authorship: Designer changes something – PCG algorithm refines it - ...

# Teleological vs. Ontogenetic





# Teleological vs. Ontogenetic



# **Teleological**

- Creates an accurate model of the result.
- PCG is the simulation of realistic processes
  - Example: A landscape is formed by simulating the process of erosion

# **Ontogenetic**

- Observe the properties of the end result
- Re-create the result without following the "natural" way to derive it
  - Example: Forming a landscape using a noise function

# **Basic Tools**



#### **Random Number Generators**

Not handled here

Regular Languages, Grammars, Automata

**Noise functions** 

**Graph algorithms** 

**Genetic algorithms, heuristics** 

### **Texture Generation**



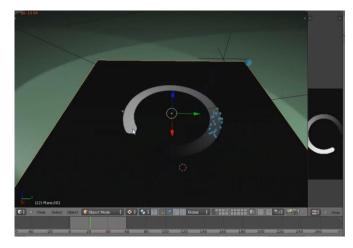
#### Generate

- Texture
- Normal Map
- Specular Map
- **-** ...

# Can be used in different systems

- Textures for objects
- Height maps
- Controlling flow or emission of particles





https://www.youtube.com/watch?v=UZGoht2vkzU

### **Texture Generation**



# **Basic Generators & Image inputs**

- Provide basic shapes and patterns
- Can insert randomness into the process
- Also image inputs to use in further steps



#### **Filters**

- Change the look of the input texture
- Enhance, blur, filter, ...
- Carry out mathematical operations

#### **Combinations**

Combine different textures





# **Texture Generation Node Networks**

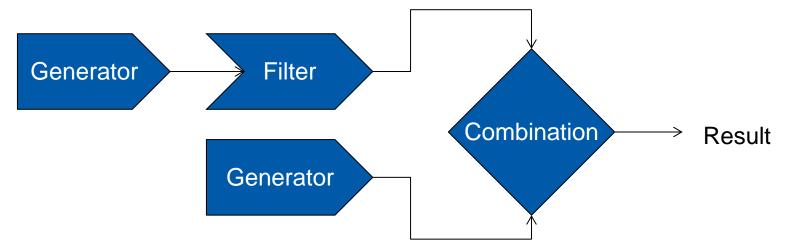


# **Combine different algorithms**

# **Basic Generators have only texture output(s)**

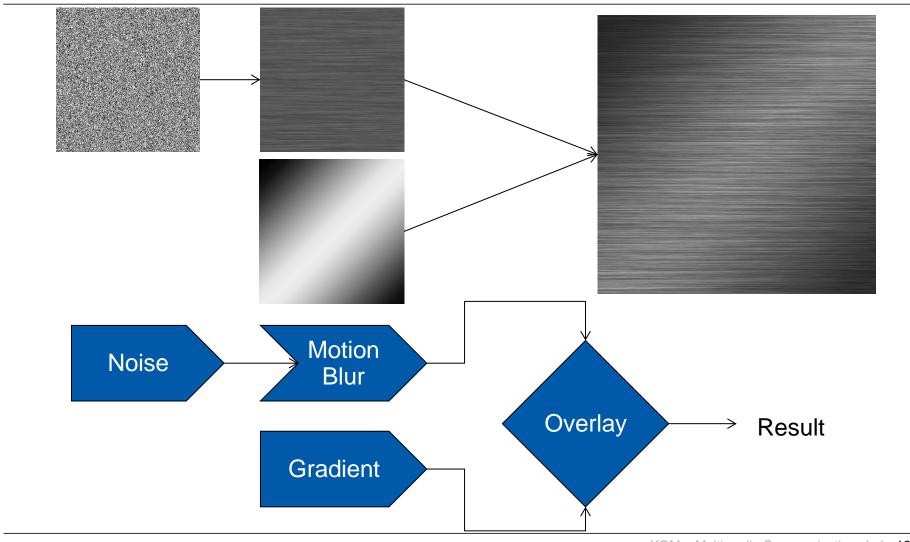
#### **Filters and Combiners have**

- One or more texture inputs
- One or more texture outputs



# **Example of networks - Metal**







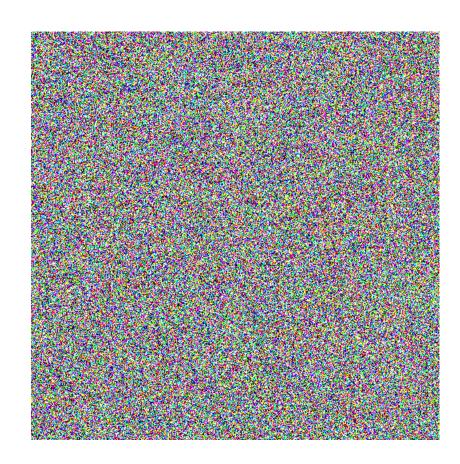
#### Random

- All colors
- Grayscale

#### **Patterns**

- Grids
- Dots/Spheres
- Jittered patterns
- Voronoi Diagram

#### **Random Noise**





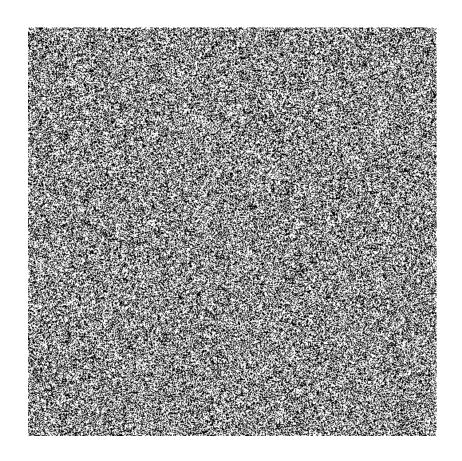
#### Random

- All colors
- Grayscale

#### **Patterns**

- Grids
- Dots/Spheres
- Jittered patterns
- Voronoi Diagram

#### **Random Noise**





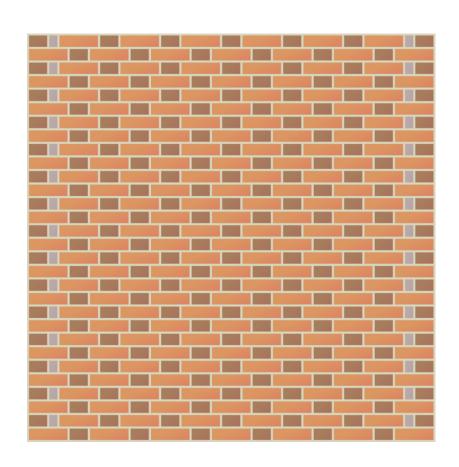
#### Random

- All colors
- Grayscale

#### **Patterns**

- Grids
- Dots/Spheres
- Jittered patterns
- Voronoi Diagram

#### **Random Noise**





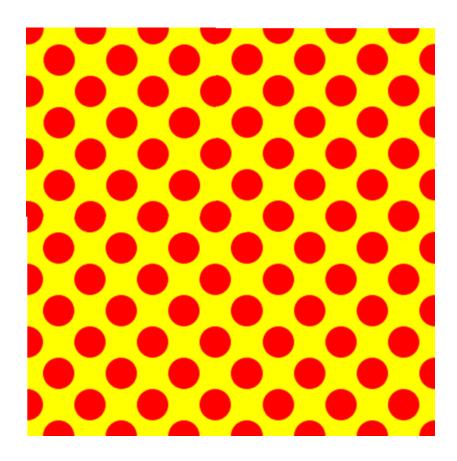
#### Random

- All colors
- Grayscale

#### **Patterns**

- Grids
- Dots/Spheres
- Jittered patterns
- Voronoi Diagram

#### **Random Noise**



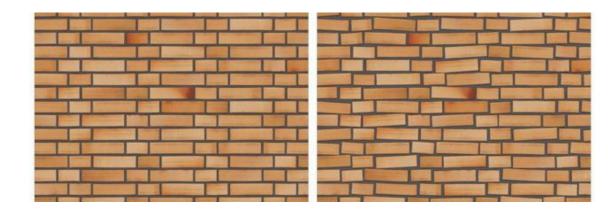


#### Random

- All colors
- Grayscale

#### **Patterns**

- Grids
- Dots/Spheres
- Jittered patterns
- Voronoi Diagram



#### **Random Noise**



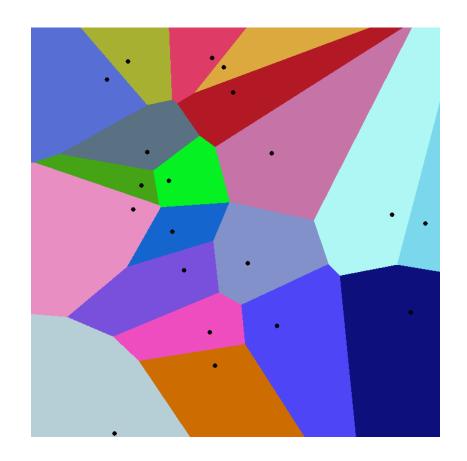
### **Random**

- All colors
- Grayscale

#### **Patterns**

- Grids
- Dots/Spheres
- Jittered patterns
- Voronoi Diagram

#### **Random Noise**





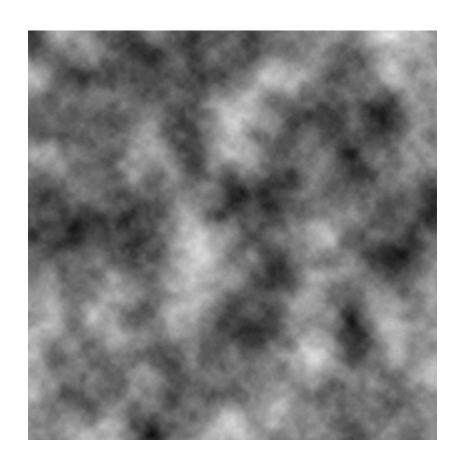
#### Random

- All colors
- Grayscale

#### **Patterns**

- Grids
- Dots/Spheres
- Jittered patterns
- Voronoi Diagram

#### **Random Noise**



### Filters - Basics



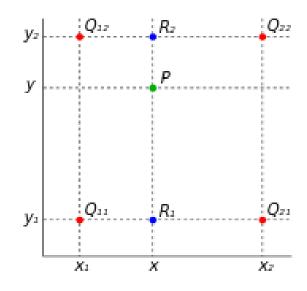
# Each pixel of the resulting image is based on one or more pixels of the input image

# Remember bilinear filtering for texture lookups

We looked up the values of 2x2 pixels to get a value for the final pixel

#### Filter kernel

Specifies the pixels we need to sample and the weights we sample them with



# **Box filter**

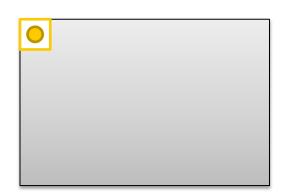


### Move a box over the image

- New pixel = Sum of original pixels \* weights
- Iterate over the image and calculate new pixels

#### Minimal Kernel size: 3x3

Size must be odd numbers (→ central pixel)



# In the following slides

- Divide by the sum of the values of the kernel → Normalization
- Alternatively, floating point numbers could be used?

# How to handle edges?

- Similar to texture lookup
- Extend the image, fill with constant color, ...

# **Box filter results**



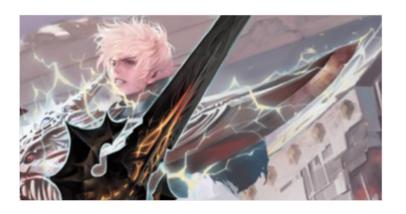
# **Unfiltered image**

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



# **Smoothing**

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



# **Box filter results**



# **Sharpening**

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$



$$\begin{bmatrix} 0 & 0 & -2 \\ 0 & 2 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$





# **Box filter results**



### **Motion Blur**

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$



# **Edge Detection**

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$



# **Gaussian Blur**



#### In one dimension

- σ is the standard deviation
- $\mu$  is not in the formula  $\rightarrow \mu = 0$

#### Calculation of the kernel

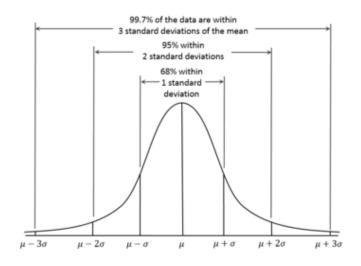
- In theory, G(x) would never be 0
- But in practice, all values further away from 0 than ~3σ will be almost 0

#### In two dimensions

- Product of two Gaussian distributions
- Can be separated into a vertical and a horizontal pass (faster)

Carrying out a box blur several times can approximate a Gaussian blur

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{x^2}{2\sigma^2}}$$



$$G(x,y) = \frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

# **Gaussian Blur – Example Kernel**



0.00000067	0.00002292	0.00019117	0.00038771	0.00019117	0.00002292	0.00000067
0.00002292	0.00078634	0.00655965	0.01330373	0.00655965	0.00078633	0.00002292
0.00019117	0.00655965	0.05472157	0.11098164	0.05472157	0.00655965	0.00019117
0.00038771	0.01330373	0.11098164	0.22508352	0.11098164	0.01330373	0.00038771
0.00019117	0.00655965	0.05472157	0.11098164	0.05472157	0.00655965	0.00019117
0.00002292	0.00078633	0.00655965	0.01330373	0.00655965	0.00078633	0.00002292
0.00000067	0.00002292	0.00019117	0.00038771	0.00019117	0.00002292	0.00000067

### **Motion Blur**



# Gaussian Blur works in all directions simultaneously

# Motion blur simulates the way a picture would be blurred on a camera's sensor

# Simple way to approximate it

- Use Gaussian Blur only in one direction
- If the blur should be rotated: Rotate the image, blur horizontally/vertically, rotate back

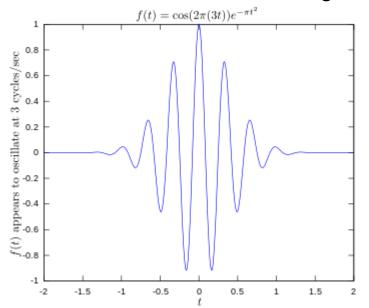
# **Filters - Convolution**

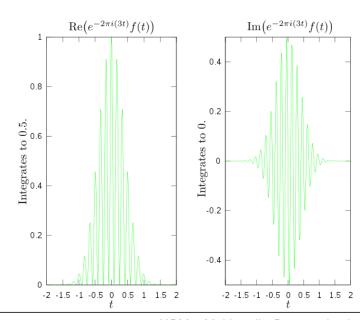


# What we are actually doing is a convolution

# (Fast) Fourier Transform

- In the regular image space, the convolution is a sum of products
- In the Fourier domain, it becomes a product
- Can be faster for filters with large kernels





# **Combinations**



# Remember the lecture on Alpha Blending Combination of source and destination pixels

#### **Modes**

- Normal blend mode
- Dissolve
- Multiply
- Screen
- Overlay
- Hard Light
- Soft Light
- Dodge and burn
- Divide
- Addition
- Subtract
- Difference
- Darken Only
- Lighten Only

Examples: http://docs.gimp.org/en/gimp-concepts-layer-modes.html

# **Example: Overlay blend mode**



Base image light → Top image become lighter
Base image dark → Top image becomes darker
a is the base image color, b the top image color

$$f(a,b) = \begin{cases} 2ab, & \text{if } a < 0.5\\ 1 - 2(1-a)(1-b), & \text{otherwise} \end{cases}$$







# **Control of PCG algorithms**



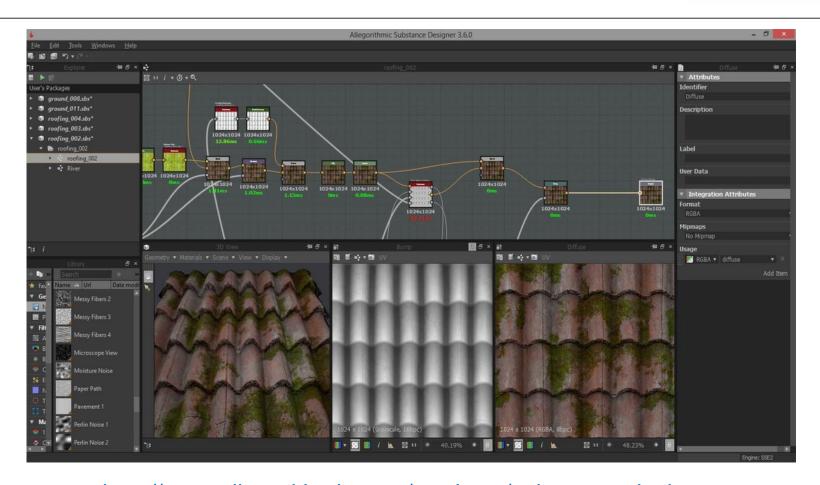
### Level of detail of controls depends on target audience

# E.g. artists on the game developer's team vs. Novice users

- How many parameters to expose?
- How many presets to provide?
- How comfortable is the configuration process?
  - Provide visual editors
  - Generate examples of the generated set

# Visual Editors – Example: Substance Designer





http://www.allegorithmic.com/products/substance-designer https://www.youtube.com/watch?v=XVk2inSEpvI#t=38

## **Voronoi Diagram**



Take a set of points C1 to Cn, "sites"

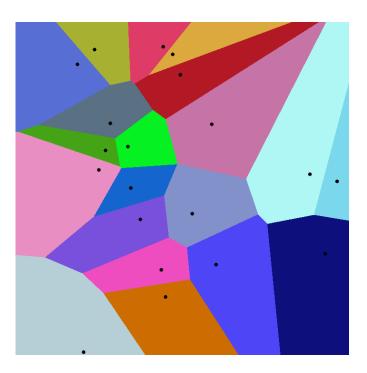
Every point Ci defines a cell such that for each point P in the cell, no other point in C lies closer to P than Ci

Regular points lead to regular patterns

Random points lead to irregular patterns

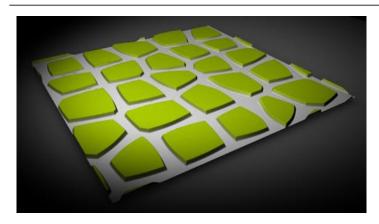
- Reptile skin
- Parcels of land
- **-** ...

(Dual to Delaunay Triangulation)

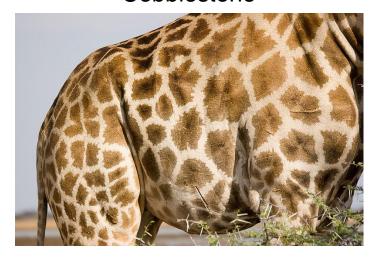


# **Voronoi Diagram – Texture examples**





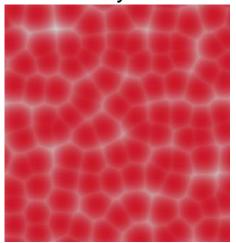
Cobblestone



Giraffe skin



Dry dirt



**Blood cells** 

## Fortune's Algorithm



### Sweep line algorithm

- We keep track of a sweep line
- → Everything on one side of the sweep line cannot be changed by the other side

#### Beach line

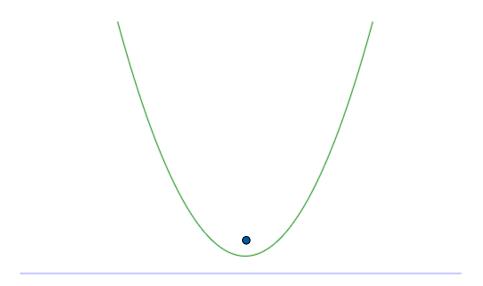
If we used only a sweep line, we could not be sure of the sweep line being "safe"

## **Beach Line**



## For a line and a point, the Voronoi diagram is a parabola

Every point inside and on the parabola is closer to the site than to the line

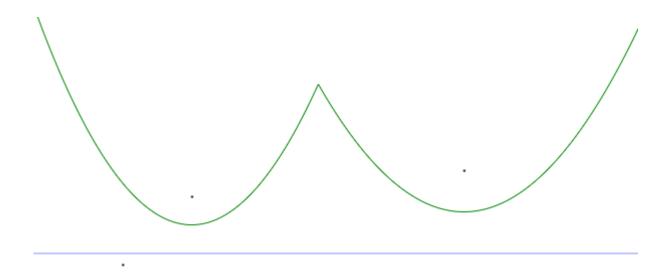


#### **Beach Line**



Several parabolas define a region in which all points are closer to the defining points than to the line

→ We can compute the Voronoi Diagram for the region above the Beach line safely

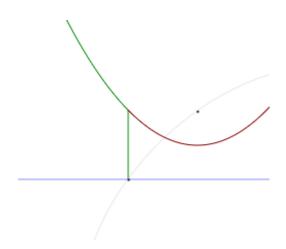


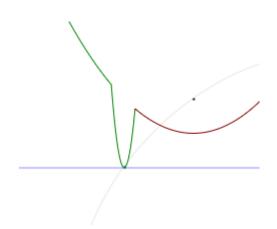
#### **Point Events**



#### Passing every new site creates a new parabola

The connecting points trace the edges of the Voronoi diagram as the sweep line moves down





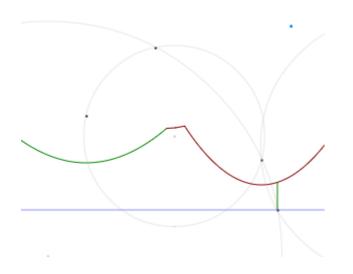
#### Circle events

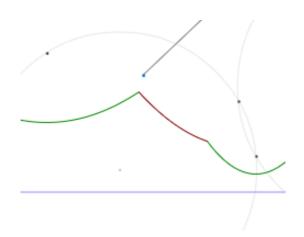


#### **Parabolas shrink**

When a parabola disappears, a vertex of the diagram is found

This point is the center of a circle that is equidistant to the three sites defining the parabola and the line





## **Examining Fortune's algorithm**



## Javascript implementation at http://www.raymondhill.net/voronoi/rhillvoronoi.html



# Voronoi Example 1 – Sir, you are being hunted



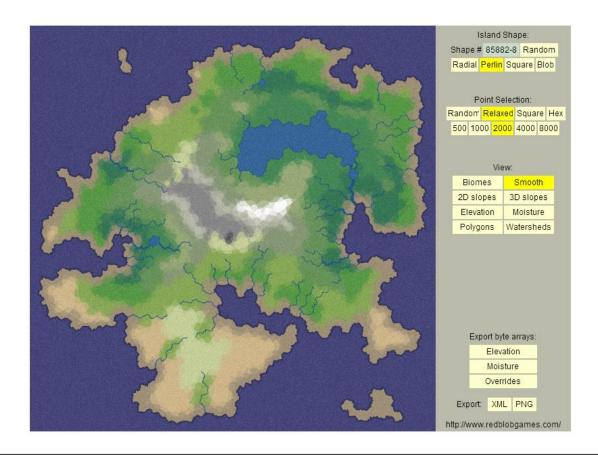
Build an "english countryside" look Using Voronoi diagram as base



## **Voronoi Example 2 – Mapgen, Amit Patel**



# http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/demo.html



## **Simplex Noise**

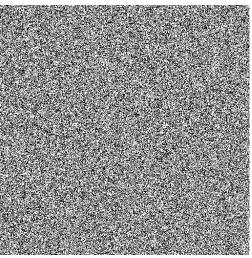


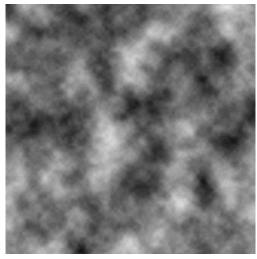
### Random pixels

- No continuity
- If we interpreted it as a 2dimensional function (heightmap), it would not work

#### **Semi-Random Noise**

- Cloud-like look
- Continuous
- Works well as a heightmap





# Perlin Noise, Simplex Noise



#### **Perlin Noise**

- Developed by Ken Perlin in 1982 for Tron
  - Won an Oscar in 1997
- Omnipresent noise generation function



#### **Simplex Noise**

- Suggested by Perlin in 2001 as a succesor to the previous noise function
- Better properties
- Scales better to higher dimensions

## **Perlin Noise**



#### **Gradient-based noise**

- Determine for each integer value
  - Function value 0
  - Pseudo-random gradient

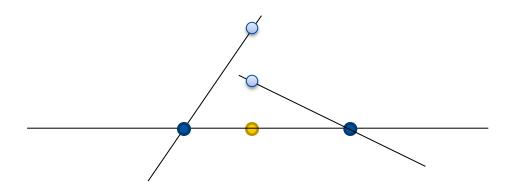


#### **Perlin Noise**



## For a given point x (2D), the result is computed by blending

- The value of the previous gradient extrapolated to point x
- The value of the next gradient extrapolated to point x



#### **Perlin Noise**

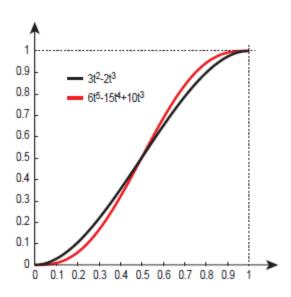


## **Blending function**

- Originally  $f(t) = 3t^2 2t^3$
- Later  $f(t) = 6t^5 15t^4 + 10t^3$

### **Purpose**

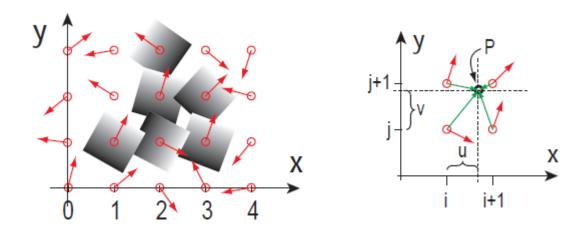
This way, the noise is also continuous at the integer positions



## Perlin Noise (2D)



For x, y, the value is interpolated between the closest integer points such that i < x < i+1, j < y < j+1



Find the unweighted contributions by using the dot product

## Perlin Noise – Dot product

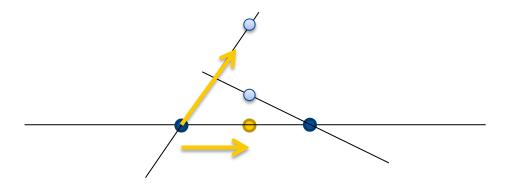


#### Can be seen better in 1D

**Gradient is the slope of the function** 

Vector towards the evaluated point is the x-Value

In this case, the dot product becomes slope \* x



## **Gradients, Computation**



#### **Choose 8-16 gradients from the unit circle**

- Generate two random numbers (x and y)
- Normalize the vector

#### To make the process repeatable

Save an array (e.g. n = 256 values) that contains a permutation of the first n integers → perm[]

## Save the gradients in an array grad[][]

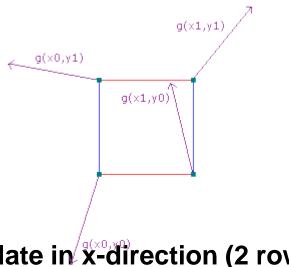
Look up using int i00 = perm[X + perm[Y] % numGradients]

# **Gradients, Computation**

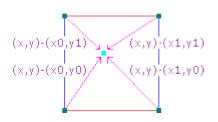


# Use the dot product to calculate the contribution of a gradient to the sample

- Gradients are defined at the grid points
- Use vectors from grid points pointing to (x, y)



Interpolate in x-direction (2 rows)
Interpolate in y-direction



## **Using Perlin Noise**



#### Normalize the noise

Divide x by width and y by height

#### Frequency

Noise = perlin(xnormalized \* frequency, ynormalized \* frequency)

### **Amplitude**

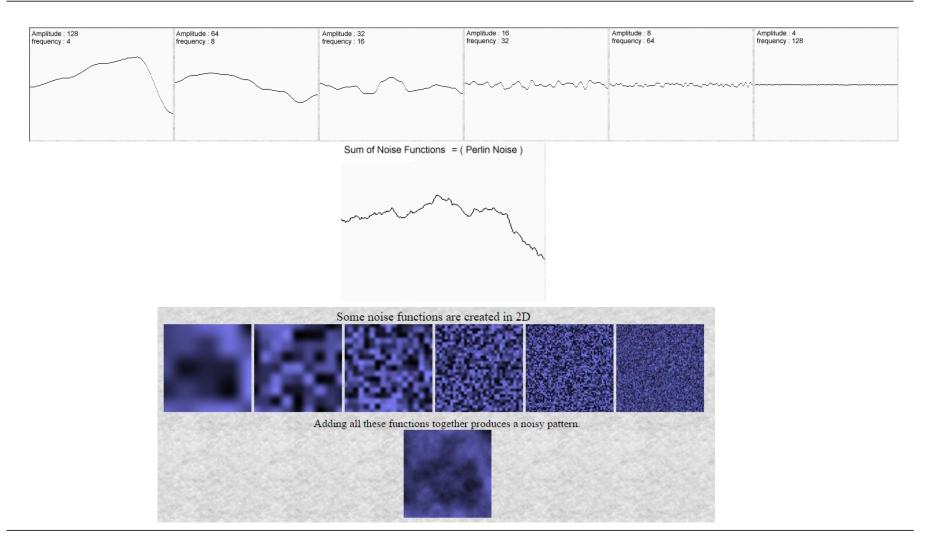
Noise = perlin(x, y) \* amplitude

#### Bring into range [0, 1]

- Noise is in [-1, 1]
- $\rightarrow$  Add 1, Divide by 2

# **Using Perlin Noise**



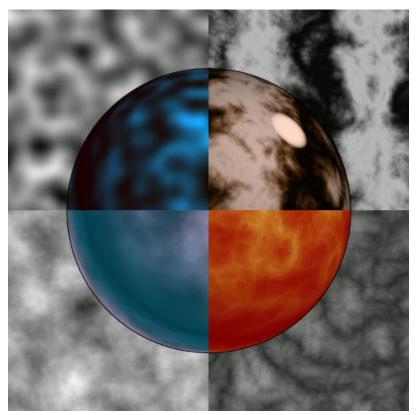


# **Using Perlin Noise**



noise

$$sin(x + |noise(p)| + \frac{1}{2} |noise(2p)| + ...)$$



 $noise(\mathbf{p}) + \frac{1}{2} noise(2\mathbf{p}) + \frac{1}{4} noise(4\mathbf{p}) \dots$ 

 $|\text{noise}(\mathbf{p})| + \frac{1}{2} |\text{noise}(2\mathbf{p})| + \frac{1}{4} |\text{noise}(4\mathbf{p})| ...$ 

#### Literature



#### **Julian Togelius**

- IT University of Copenhagen
- http://julian.togelius.com/

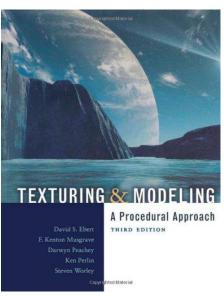
## **Procedural Content Generation in Games - A textbook and an** overview of current research

Available for free at http://pcgbook.com/

#### **PCG Wiki**

http://pcg.wikidot.com

Ebert, Musgrave, Peacheay, Perlin, Worley: Texturing & Modeling – A procedural approach



## Vegetation



One of the oldest fields of procedural content generation in computer graphics

Very much info available at: http://algorithmicbotany.org/

#### Well suited for generation

- Based on natural processes
- Complexity makes the shapes look realistic
- Can be found be examining how nature handles growth



## L-Systems



#### **Prucinciewyz Lindenmayer**

 Book "The algorithmic beauty of plants" is available for free at virtualbotany

#### L-System

- Non-deterministic formal grammar
- Follows the rules of growth

#### Two growth processes

- Winter: Pruning of excess sprouts
- Spring: Creation of new sprouts
- Sprouts become branches over time



## L-System



$$G = (V, \omega, P)$$

- **V** (the *alphabet*) is a set of symbols containing elements that can be replaced (*variables*)
- **ω** (start, axiom or initiator) is a string of symbols from **V** defining the initial state of the system
- **P** is a set of *production rules* or *productions* defining the way variables can be replaced with combinations of constants and other variables.

A production consists of two strings, the *predecessor* and the *successor*. For any symbol A in V which does not appear on the left hand side of a production in P, the identity production  $A \rightarrow A$  is assumed; these symbols are called *constants* or *terminals*.

## L-System Example



**Example: Fractal plant** 

variables: X F

constants: + - []

start: X

Rules:  $(X \rightarrow F-[[X]+X]+F[+FX]-X)$ ,

 $(F \rightarrow FF)$ 

angle: 25°

F: "draw forward"

-: "turn left 25°"

+: "turn right 25°"

X: Intermediate Symbol

[ push position and angle

] pop position and angle



## L-Systems



# The resulting string is transformed into a mesh using a turtle algorithm

#### Small branches and foliage as billboards

Placement as determined by the system

#### Movement of the tree

- Simple animation where the top of the tree sways more than the bottom
- More physically correct: Simulate individual branches

# Programming – Virtual classes, interfaces



#### So far, we have seldomly used virtual classes or inheritance

#### Overhead that this incurs

- Virtual function table
- Inlining not possible

## **Advantages**

- Better coding style
- Build interfaces that other programmers can implement

#### Virtual functions in C++



#### Making the function

- Add the virtual keyword to a function declaration
- class A {
  - virtual void foo(int a);
- }

### **Deriving from the class**

Usually use class B: public A { };

#### **Overriding methods**

- Provide the function in the inheriting class
- class B: public A {
  - virtual void foo(int a);
- **•** }

# **Casting**



## **Implicit**

- A\* a;
- B b;
- a = &b;

## **Explicit**

- A\* a;
- B b;
- a = (A\*) &b;

## **Casting**



## dynamic\_cast (requires run-time type information RTTI)

- dynamic\_cast<type>(pointer);
- Returns nullptr if the cast is not possible
- Can upcast and downcast

#### static\_cast

- static\_cast<type>(pointer);
- No type checks

#### reinterpret\_cast

- All combinations
- Also cast integers to pointers

## **Abstract classes/functions**



No dedicated keyword

virtual void abstract\_function() = 0

**Build interfaces from these** 

## **Constructors, Virtual destructors**



#### Constructors

- No-parameter constructor called automatically
- Constructors with parameters must be called in the initialization list

```
class Sub : public Base
    Sub(int x, int y): Base(x), member(y)
    Type member;
 - };
```

#### Virtual destructors

Need to be virtual to be callable by a pointer to the base class

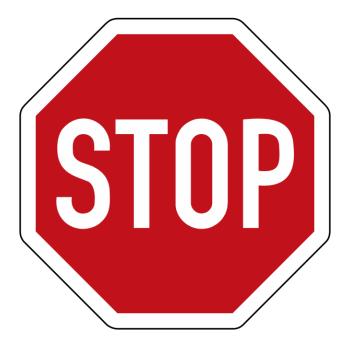
## **Multiple Inheritance**



#### Use with caution

## Can make sense if you know what you are doing

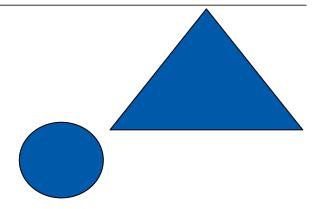
■ E.g. re-produce interfaces from other programming languages





#### 1.1 Separating Axes test

The test did not really depend on this axis



#### 1.2 Collision Basis

- Use the formulae from the slide
- One border case: What if the collision normal was parallel to the chosen axis?

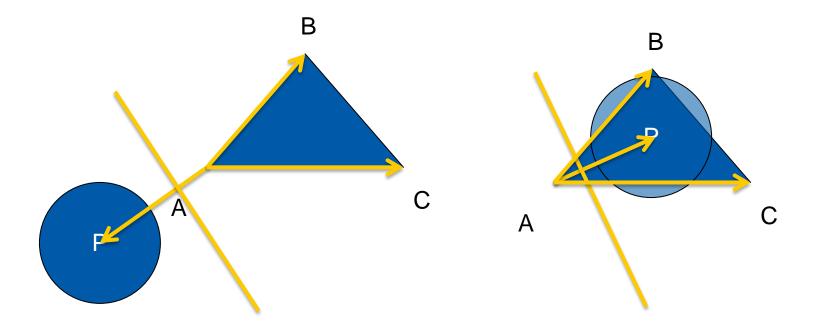
## 1.3 Depends on your theoretical solution

#### **Theoretical Task**

We will provide links/descriptions to the tests you found in the solution

# **Separating Axes Test**





#### **Exercise 10 – Procedural Content Generation**

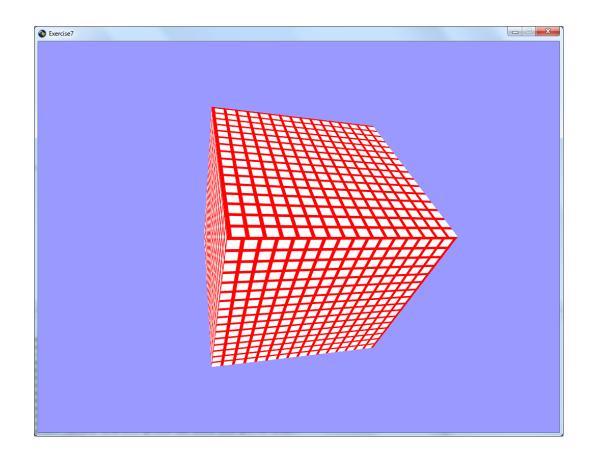


### Build a texture generator of your choice

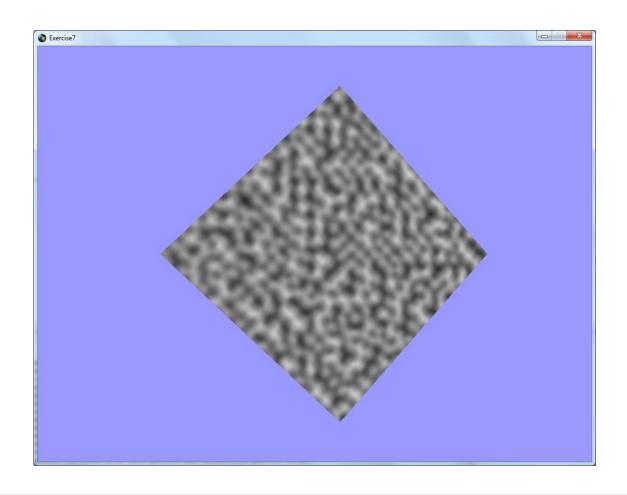
If you cannot think of one, you can use the ones from the lecture

- 1.1 Implement Perlin Noise
- 1.2 Implement Gaussian Blur
- 1.3 Implement the "Overlay" combine mode
- 1.4 Create your own network

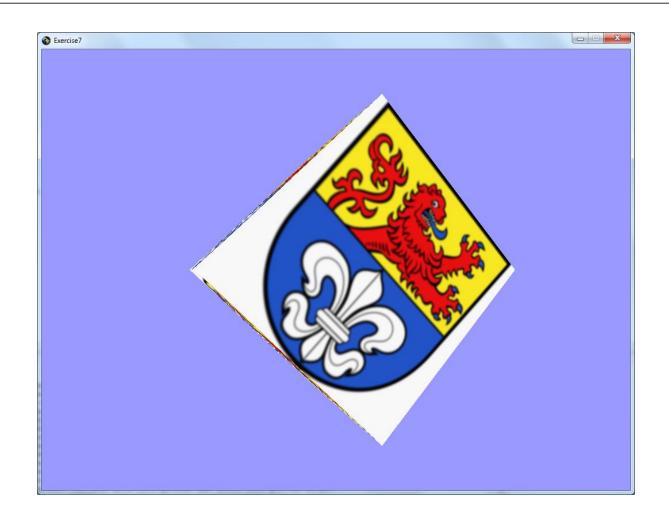




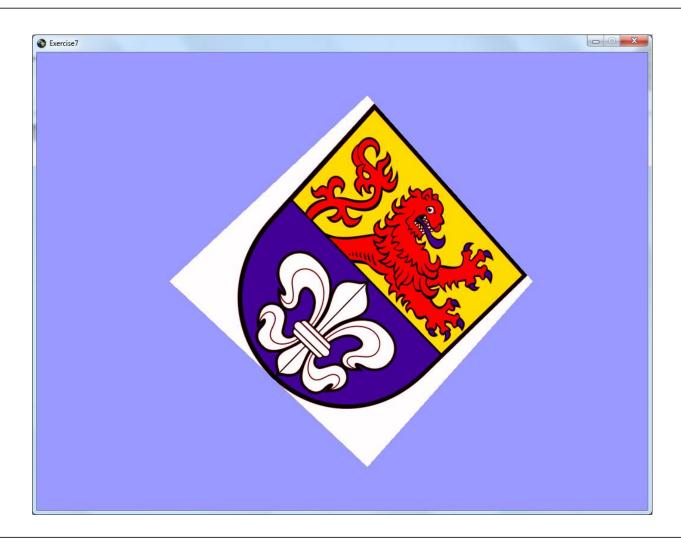




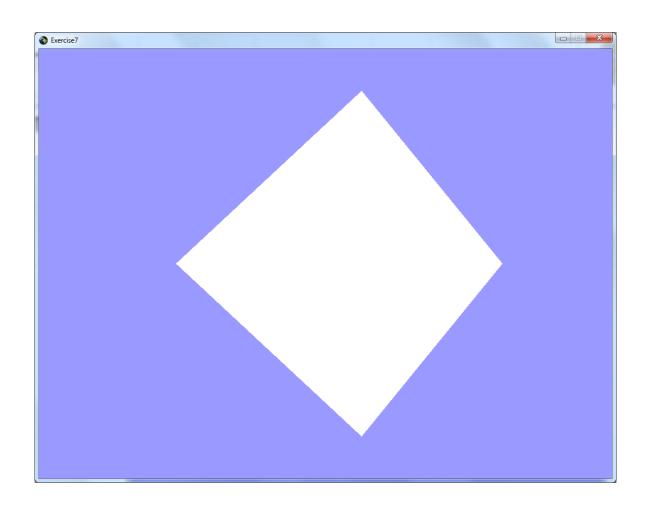












#### **Exercise 10 – Theoretical Exercises**



#### 2.1 Voronoi Diagram

- Imagine a set of points (e.g. cities in an RTS game)
- How can you visualize the "border" of the area controlled by the cities by making use of a Voronoi Diagram?

#### 2.2 Analyze an edge detect kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

## 2.3 Analye the properties of the Gaussian distribution

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{x^2}{2\sigma^2}}$$



## **Questions & Contact**







Department of Electrical Engineering and Information Technology Multimedia Communications Lab - KOM



Dr.-Ing. Florian Mehm

Florian.Mehm@KOM.tu-darmstadt.de Rundeturmstr. 10 64283 Darmstadt Germany

Phone +49 (0) 6151/166885 Fax +49 (0) 6151/166152 www.kom.tu-darmstadt.de



game-technology@kom.tu-darmstadt.de