# GED Practical Course

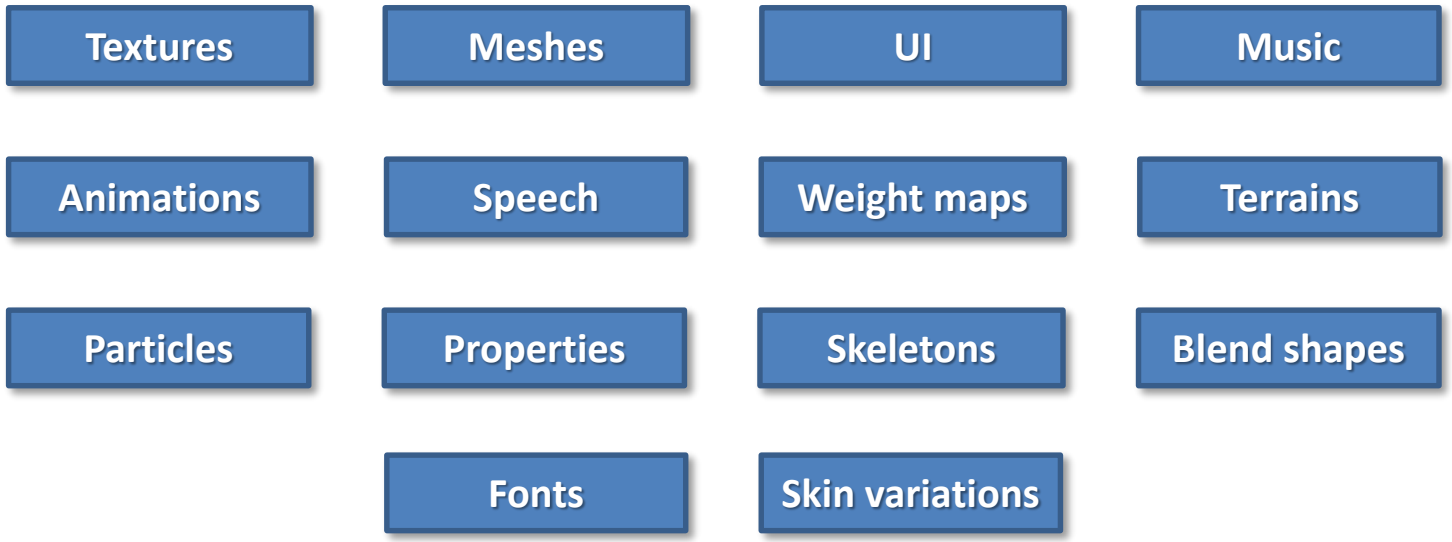


computer graphics & visualization

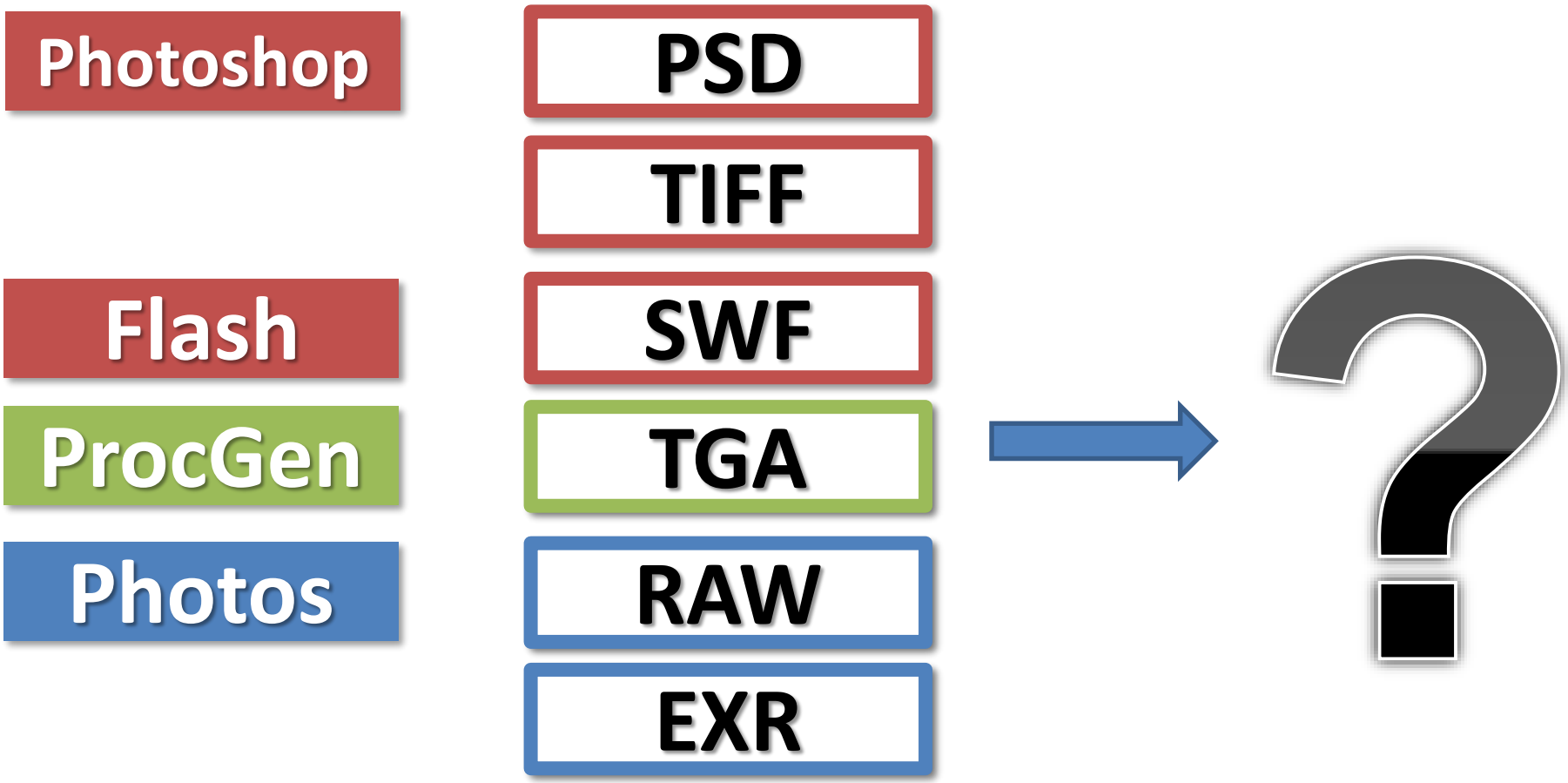
# A quick journey: game content creation

# Art integration

- Modern games require a large number of „assets”



# Art integration



# Art integration

- “Authoring” formats typically require a lot of memory
  - Code for loading those might be slow and complex
  - Prone to errors

- During runtime: Different requirements
  - Level of detail for textures / geometry dependent on target platform (PC, console, mobile device)
  - Preprocess the data as far as possible
    - Optimal runtime performance
    - Throw away unneeded data
    - Maybe compress files for faster loading

# Art integration

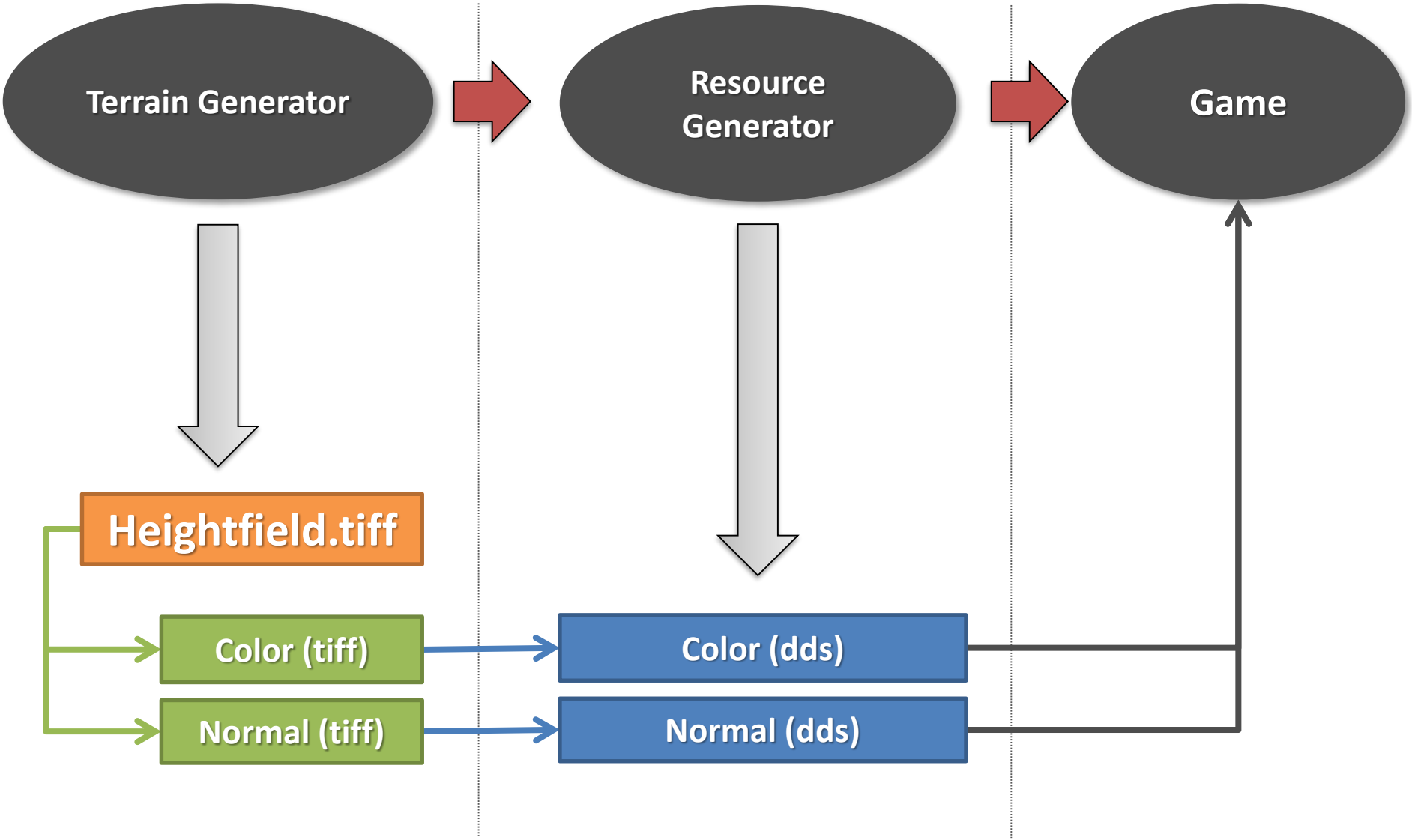
# Art Pipeline / Content Processing

- What makes a good art pipeline?
  - Early asset validation
  - Dependency tracking
  - Easy to insert new stages
  - Easy to customize
  - Efficient, both in storage and processing time
  - **Fully automated**
    - No manual user intervention
    - „One click“ execution
    - Extensive logging and error tracking facilities

# Assignment 3

- Assignment 3: Texture Generation
  - Replace `GEDUtils::TextureGenerator` with own code
    - Normal map creation
    - Diffuse color map creation
  - Preprocess content for in-game usage
    - Downsize heightfield
    - Convert textures
    - Automation of this process

# Art pipeline



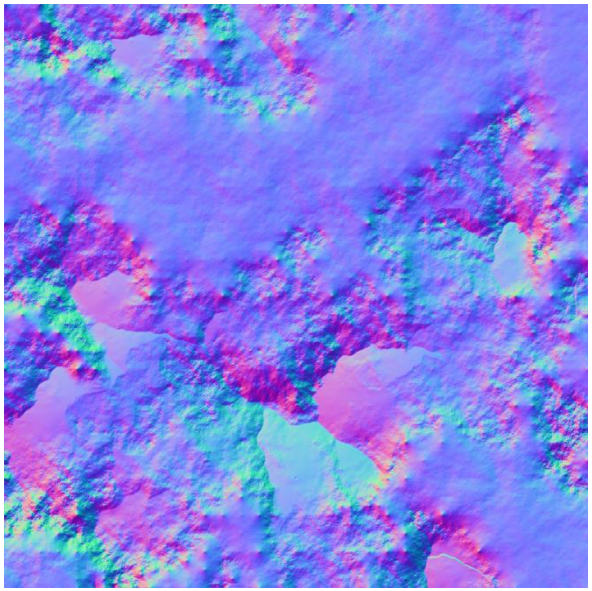
# Art Pipeline

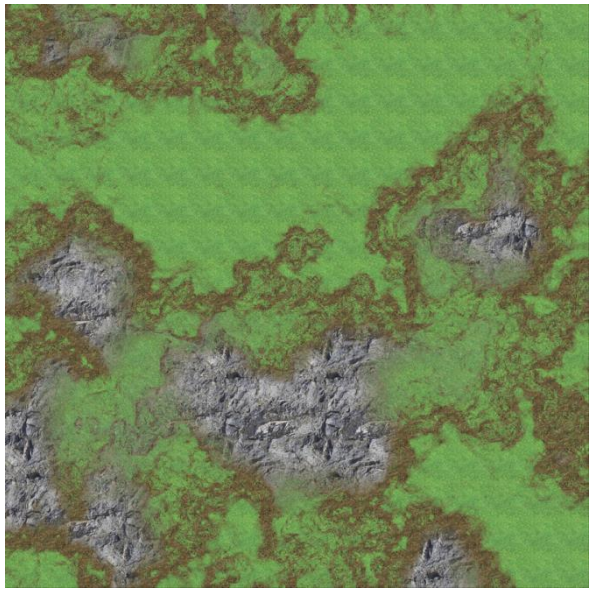
1) Create height field
2) Calculate normals ($4096^2$)
3) Save normals to .TIFF
4) Load material textures (Grass, Rock, etc.)
5) Blend material textures
6) Save color texture to .TIFF
7) Downsize heightfield for use in-game
8) Save heightfield

TODO this week

# TerrainGenerator

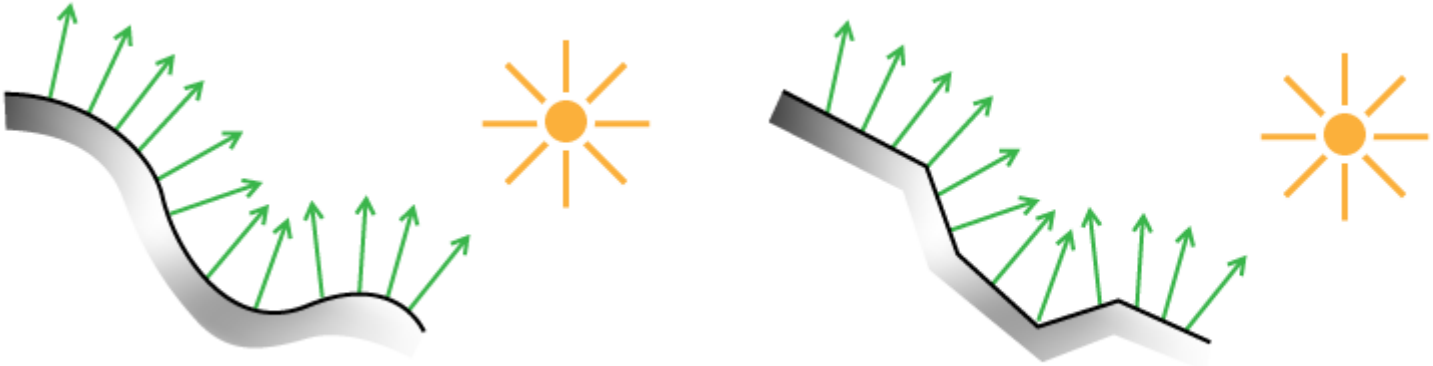
- Current output: 2D heightfield
  - Additional input
    - 4 color textures
- Additional output: Normal-Map, Color-Map
  - Until now, `TextureGenerator` handled this for you!

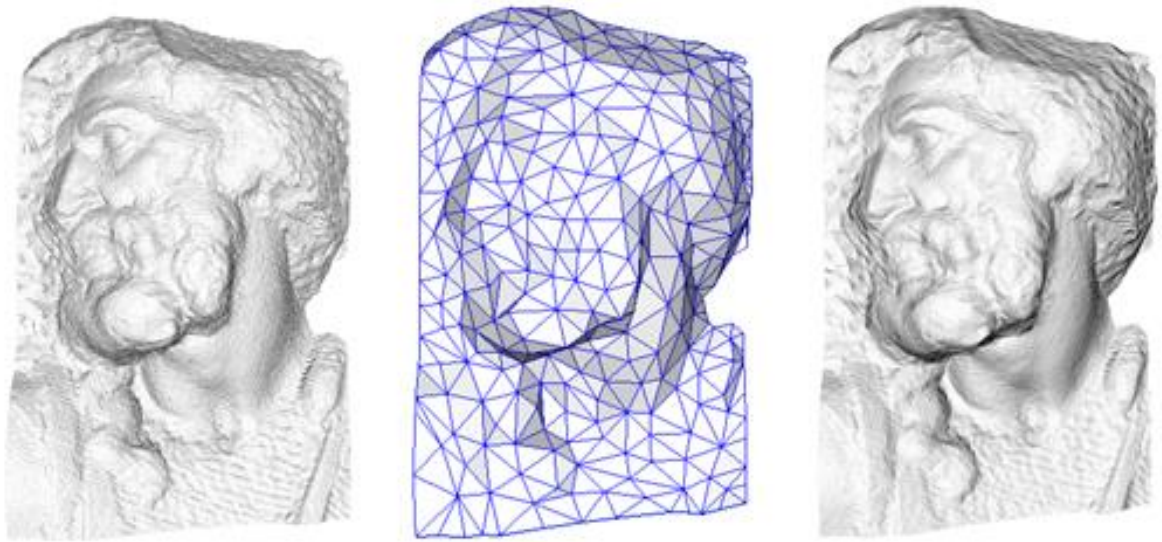




# Normal Mapping

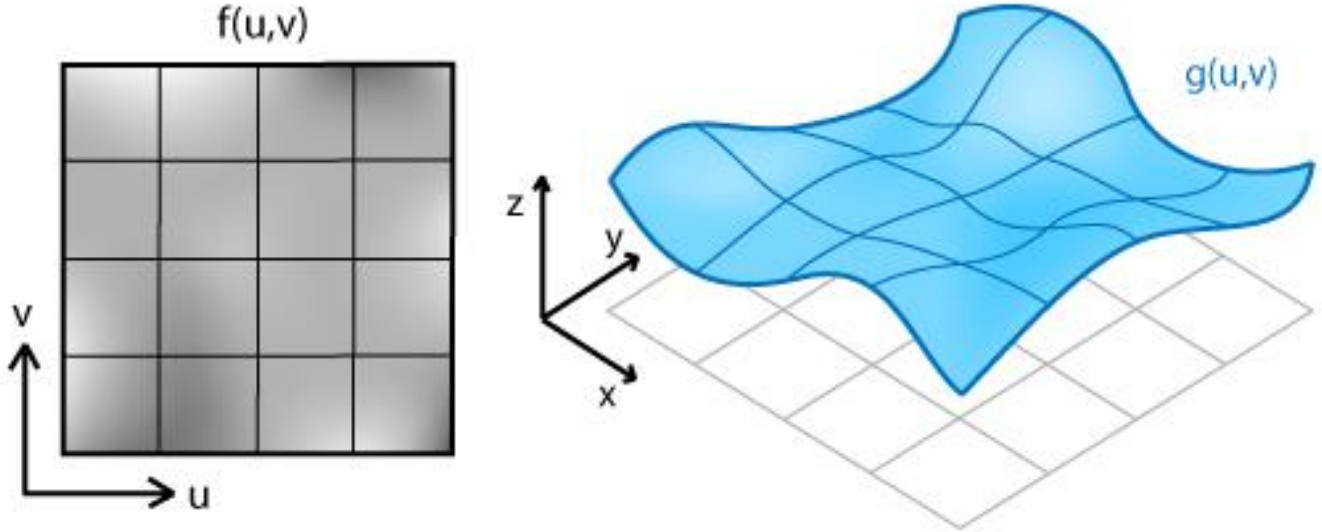
- Normal mapping





# Normal Mapping

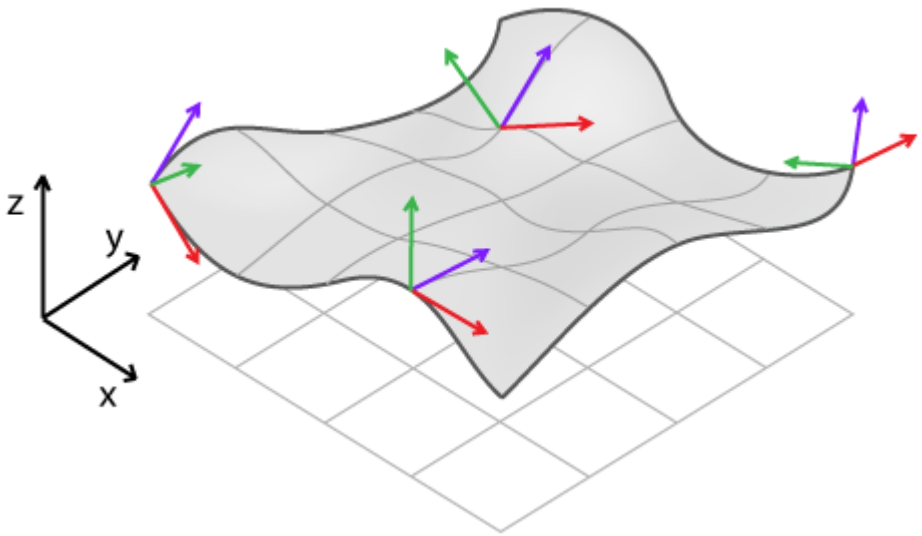
- Normal mapping for 2D height fields
  - http://acko.net/blog/making-worlds-3-thats-no-moon/




$$g(u,v) \rightarrow (x,y,z) : \begin{cases} x = u \\ y = v \\ z = f(u,v) \end{cases}$$

# Normal Mapping

- Normal mapping for 2D height fields




**Approximation using central differences:**

(assuming: du, dv = 1)

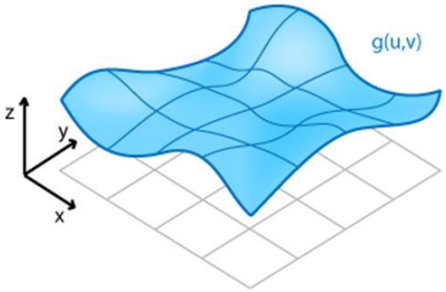
$$\frac{\partial f}{\partial u} = (map[u+1][v] - map[u-1][v])/2$$

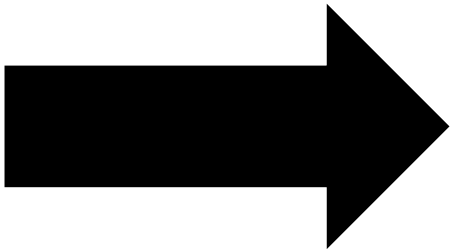
$$\frac{\partial f}{\partial v} = (map[u][v+1] - map[u][v-1])/2$$

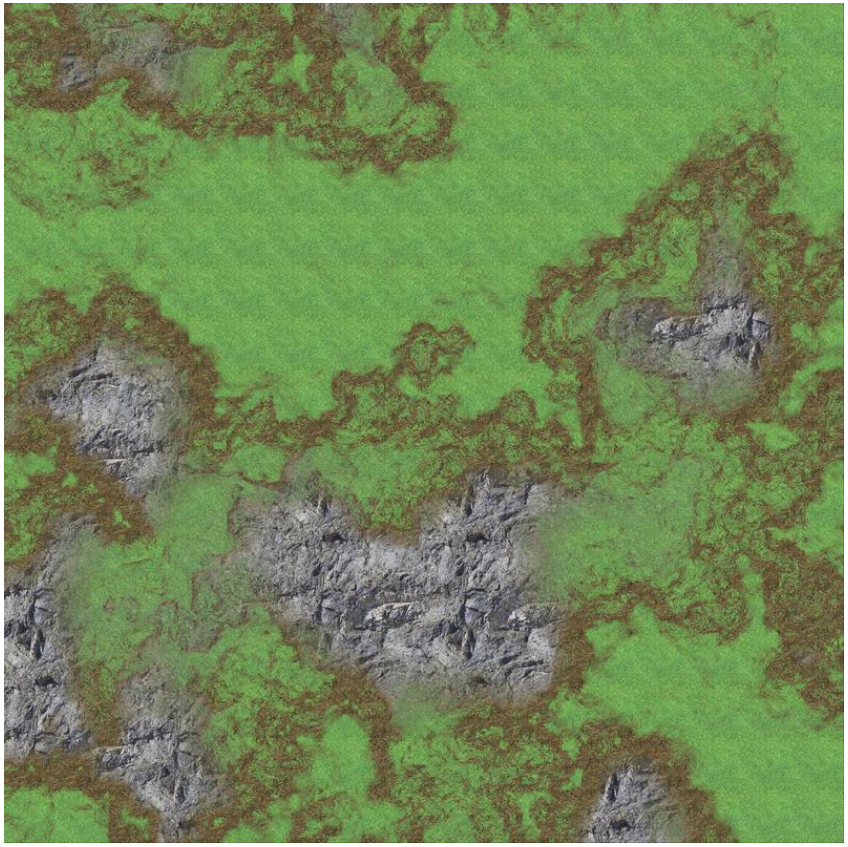
$$\mathbf{t_u} \times \mathbf{t_v} = \mathbf{n} = \begin{bmatrix} -\frac{\partial f}{\partial u} & -\frac{\partial f}{\partial v} & 1 \end{bmatrix}$$

- Caution! Terrain has different units for x / y and z
  - x/y are in pixels
  - z is in [0,1]
- Normalize: $\mathbf{n} \leftarrow \frac{\mathbf{n}}{\|\mathbf{n}\|}$

# Color Texture Creation



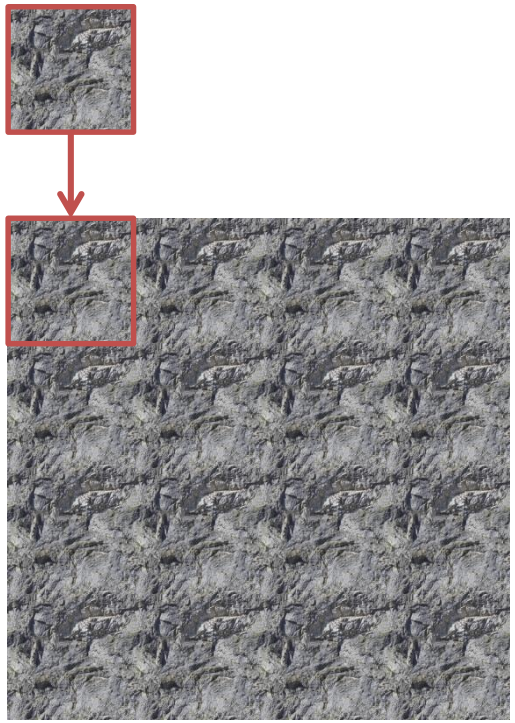





# Seamless Texturing

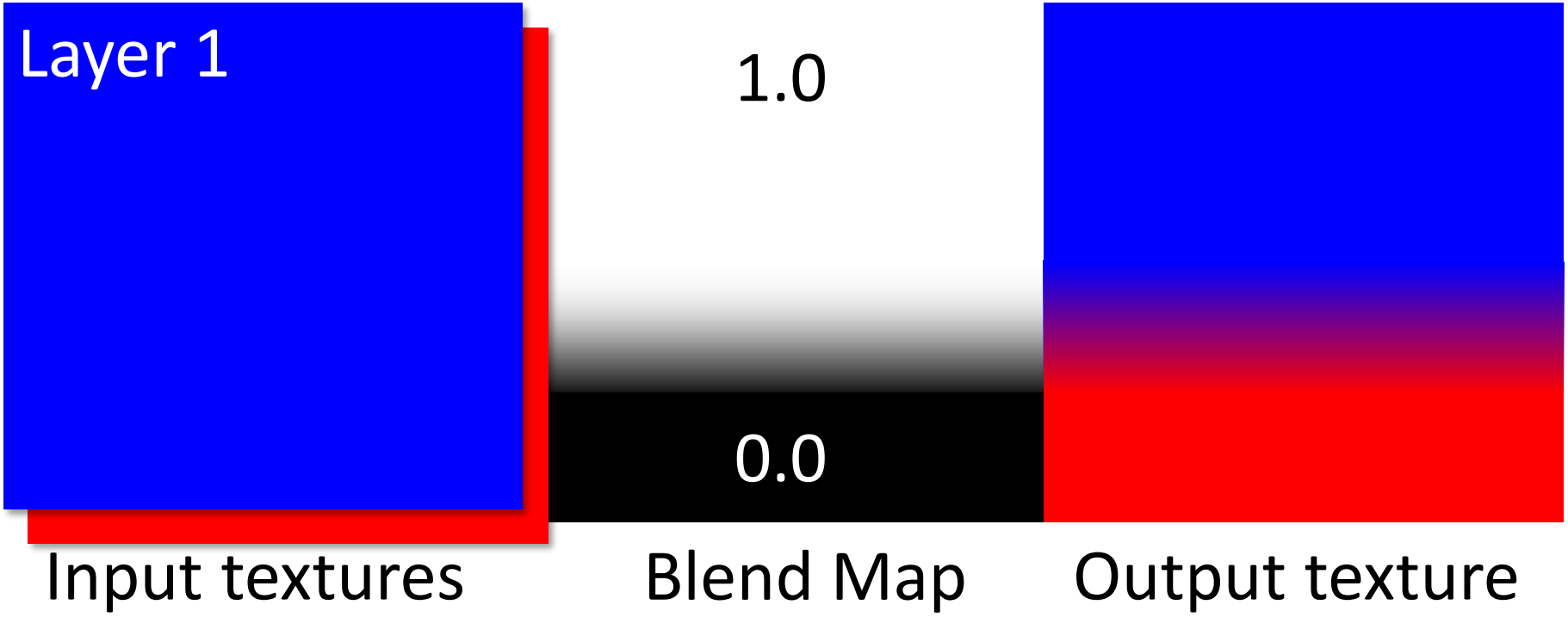
- Input textures are smaller than the desired terrain resolution
- But: Textures are tileable!
  - Left side matches right side
  - Top side matches bottom side
- Straight forward implementation:

```
color Texture::getColorTiled(int u, int v) {
    // u,v are texture coordinates
    return this->getColor(
        u % this->res.u, v % this->res.v);
}
```



# Texture Blending

- Alpha-Blending for two layers:
  - Weight for layer 1: $\alpha$
  - Weight for layer 2: $1 - \alpha$
  - $\alpha$ has to be in [0,1]



# Texture Blending

- *Alpha-Blending for N layers*: Layer $i$ is blended over layer $i-1$:
  - $C_0 = c_0 \qquad C_i = \alpha_i * c_i + (1 - \alpha_i) * C_{i-1},\ i \in [1, N-1]$
- So, for 4 layers:
  - $C_3 = \alpha_3 * c_3 + (1 - \alpha_3) * (\alpha_2 * c_2 + (1 - \alpha_2) * (\alpha_1 * c_1 + (1 - \alpha_1) * c_0))$
  - Lowest layer is fully opaque, does not need to be blended
- Order is important!
  - $c_0$ below $c_1$ below $c_2$ below $c_3$
  - Each blend weight has to be in [0,1]
  - Sum does not neccessarily have to be 1!

# Texture Blending

- Textures are generated depending on the heightfield
  - Based on height and slope of each pixel
- For example:

| Slope / Height | Low | High |
|---|---|---|
| Steep | Dirt ($\alpha_1$)  | Rock ($\alpha_3$)  |
| Flat | Grass  | Pebbles ($\alpha_2$)  |

- Slope determined by the normal (i.e.: $1 - \boldsymbol{n}.z$)
- Simple idea for calculating blend weights:

```
void calcAlphas(float height, float slope, float& alpha1, float& alpha2, float& alpha3) {
	alpha1 = (1-height) * slope;
	alpha2 = height;
	alpha3 = height * slope;
} // Problem: Very smooth blending, can you improve this?
```

# Normal Calculation

- Calculate the normals for each pixel using central differences
  - Store this in an additional 2D array with 3 floats per normal!
  - You may want to use a `struct` for these 3 floats…
- Save to TIFF
  - Use the provided `SimpleImage` class
  - You need to save RGB instead of a single grey value, of course!
  - `SimpleImage` expects values in the range [0;1]
    - Will be automatically converted into RGB when saving as TIFF
    - Caution! Your normal values are in [-1;1]! Convert them accordingly!

# Texture Blending

- Use `SimpleImage` to load the color textures
  - `GetPixel(x, y, r, g, b)`
  - `SetPixel(x, y, r, g, b)`

- Encapsulate blend weight calculation into a function
  - Easier to tweak
  - Simply call this for each pixel with a given height / slope

# Automation

- Automation is important
- „One-click“ builds
- Only the following steps should have to be performed to create your game content:
  - SVN Checkout
  - Open solution
  - Set configuration to RELEASE
  - Set startup project if neccessary
  - Run

# Automation

- Your “Art Pipeline” is a custom Visual Studio project
  - No code, only a makefile
  - Makefile will call the required tools
  - See assignment for details

- Project dependencies will keep everything up to date
  - “If this project is built, also build the other project if necessary”
  - Automatically creates all resources when your game is built lateron
    - Automatically builds your TerrainGenerator when resources need to be created

# TexConv

- TexConv
  - Command line tool for DirectXTex
  - Included in the template solution!
  - Converts textures into a GPU friendly format
    - MipMap generation
    - Format conversions
    - Optional compression
  - Output: .dds file

# NMake

- “Building” the NMake project will execute all commands in “Build command line”
  - In our case: Calls all tools of the art pipeline
  - Right-Click project -> “Build”

- “Cleaning” the NMake project will execute all commands in “Clean command line”
  - Right-Click project -> “Clean”
  - Deletes heightfield, color map and normal map (tiff and dds)

# NMake

- NMake example:

```
"$(OutDir)TerrainGenerator.exe" -r 512 -o_height
"$(OutDir)terrain_height.tiff" -o_color "$(IntDir)terrain_color.tiff" -
o_normal "$(IntDir)terrain_normal.tiff"

"$(OutDir)texconv.exe" ... <more parameters>
```

- `$(OutDir)` is an environment variable defined in Visual Studio
  - Contains the output directory of the current project (your .exe file is written to this location)
  - Determined by the project settings!
- `$(SolutionDir)` contains the directory of your .sln file
- `$(IntDir)` contains temporary build files

# Questions?