CMSC 427: Chapter 6
Texturing and Surface Detail

Reading: Chapt 11 in Shirley
Overview:
- Texture mapping
- Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
- Procedural textures and Perlin noise

Elements of Realistic Rendering

Realism in Rendering:
- Perspective projection
- Illumination and shading
- Texture mapping and surface detail
- Hidden surface removal
- Color
- Reflection and other effects
Texture and Surface Detail

We have seen how to provide color to objects using:
- **Solid colors**: using `glColor()`.  
- **Lighting and shading**: through various lighting and shading models.

Today we consider how to add realism through **surface detail**. 
Examples:
- **Natural surfaces**: stone, wood, gravel, grass.  
- **Printing and painting**: printed labels, billboards, newspapers.  
- **Clothing and fabric**: woven and printed patterns, upholstery.

Image Texturing

![Image courtesy, Foley, van Dam, Feiner, Hughes](image_url)
Overview

• The texture-mapping process

• Texture mapping in OpenGL

• Other types of maps: Bump, Displacement, Environment

• Procedural textures

• Perlin noise

Texture and Surface Detail

Texture Mapping:
- Conceptually, a texture is an \textit{image pasted on the surface} of an object.

Textures as Micro-Geometry:
- Textures provide a way to model \textit{repetitive and high-resolution} color and geometric features, without complex geometry.

Generalizations: There are elements other than color that can be mapped onto a surface.
- \textit{surface normals} (used for lighting computations)
- \textit{bumps} and \textit{surface displacements}.
- \textit{environments} (used to “fake” reflective surfaces)
- \textit{shadows}, ...
Texture Mapping Process

Texture wrapping function: Maps from texture to object space.

Inverse wrapping function: Maps the other way. This is actually what we need in texture mapping—which texel corresponds to a given surface point.

Surface Parameterization

Parameterization:
In order to simplify the process of mapping a 3-d surface point to a 2-d texture point, it is common to first represent the surface using a 2-d parameterization.

Two-step Inverse Mapping:
3-d object space: (x, y, z)
→ 2-d parameter space: (u, v)
→ 2-d texture space: (s, t)
Example: Texture Mapping a Sphere

Example: Mapping a sphere.
Consider a sphere of radius \( r \), centered at the origin.
Consider a point with coordinates \((x, y, z)\) on this sphere.
What are the corresponding \((s, t)\) coordinates?

Spherical Coordinates:
Consider the vector \( w = (x, y, z) \) with respect to the sphere center.
Let \( r = \| w \| = \sqrt{x^2 + y^2 + z^2} \).
Let \( \phi \) be the angle with respect to the z-axis.
Let \( \theta \) be the angle with respect to the x,y-plane.
Then:
\[
\begin{align*}
  z(\theta, \phi) &= r \cdot \cos \phi \\
  x(\theta, \phi) &= r \cdot \cos \theta \sin \phi \\
  y(\theta, \phi) &= r \cdot \sin \theta \sin \phi,
\end{align*}
\]
where \( r = \sqrt{x^2 + y^2 + z^2} \).

Surface Parameterization: \((\theta, \phi)\), where
\[
\begin{align*}
  \theta(x,y,z) &= \arctan(y/x), \text{ where } 0 \leq \theta \leq 2\pi \text{ and} \\
  \phi(x,y,z) &= \arccos(z/r), \text{ where } 0 \leq \phi \leq \pi.
\end{align*}
\]
These encode something like the longitude and latitude of the point.

Inverse Wrapping Function: Assuming that texture coordinate range is \( 0 \leq s, t \leq 1 \), let
\[
\begin{align*}
  s(x,y,z) &= \theta(x,y,z)/2\pi = (\arctan(y/x))/2\pi, \\
  t(x,y,z) &= \phi(x,y,z)/\pi = (\arctan(z/r))/\pi.
\end{align*}
\]
Example 2: Texture Mapping a Cylinder

Consider mapping a label around a cylinder. Assume that \((s, t)\) space has been normalized to \(0 \leq s, t \leq 1\).

Texture Mapping a Cylinder (cont)

Parametric Equation of a Cylinder: of radius \(r\), with base centered at the origin, and height \(H\).
\[
\begin{align*}
x &= r \cdot \cos \theta, & \text{for } 0 \leq \theta \leq 2\pi \\
y &= r \cdot \sin \theta, & \text{for } 0 \leq \theta \leq 2\pi \\
z &= h, & \text{for } 0 \leq h \leq H.
\end{align*}
\]

Surface Parameterization: \((\theta, h)\), where
\[
\begin{align*}
\theta(x,y,z) &= \arctan(y/x), & \text{where } 0 \leq \theta \leq 2\pi \\
h(x,y,z) &= z, & \text{where } 0 \leq h \leq H.
\end{align*}
\]

Inverse wrapping function: Assuming that texture coordinate range is \(0 \leq s, t \leq 1\), let:
\[
\begin{align*}
s(x, y, z) &= \theta(x,y,z))/2\pi = (\arctan(y/x))/2\pi, \\
t(x, y, z) &= h(x,y,z)/H = z/H.
\end{align*}
\]

Can use \(\text{atan2}(y,x)\) to compute this.
General Texture Mapping

**General Approach:** need a 2-d parameterization of a 3-d surface.

- **Trivial** for some simple objects: sphere, cylinder, cone, cube, ...
- **Difficult** for complex objects: described by a surface mesh.

**Current schemes involve:**
- Two-stage mapping, or
- Converting a 3-d surface into an atlas of different parameterizations.

**Two-stage Texture Mapping** by Bier and Sloan (1986).

- **S-Map:** Maps from 2-d texture space to an intermediate (simple) 3-d surface (cylinder, sphere, cube, ...

\[ T(s, t) \rightarrow S(r, \theta, \phi) \]

- **O-Map:** Maps from 3-d intermediate space to the final (possibly complex) 3-d object surface:

\[ S(r, \theta, \phi) \rightarrow O(x, y, z) \]

---

**Two-Stage Texture Mapping**

**Spherical Mapping:**

**Cylindrical Mapping:**

Images courtesy, David Ebert, Purdue

---

Chapter 6, Slide 13
Copyright © D. M. Mount and A. Varshney
Overview

• The texture-mapping process

Texture mapping in OpenGL

• Other types of maps: Bump, Displacement, Environment

• Procedural textures

• Perlin noise

Texture Mapping in OpenGL

Basic Steps:
- Create and specify a texture object:
  - Create a new texture object. (OpenGL provides identifier.)
  - Specify the texture image. (Provide an image.)
  - Specify how texture has to be applied for each pixel.
- Enable texture mapping.
- Draw the textured polygons.
  - Identify the texture to be used.
  - Specify texture coordinates with vertices.
- Disable texture mapping (when returning to normal drawing).

Timing:
- OpenGL requires that a graphics window exists before some texture-related commands can be issued. Issue texture commands only after first reshape or display callback is generated.
Create Texture Object(s)

`glGenTextures ( GLsizei n, GLuint* textureIDs );`
- Requests that \( n \) new texture objects be created.
- Returns \( n \) new texture IDs in \( \text{textureIDs} \).
- Each texture ID is an integer greater than 0.

`glBindTexture ( GLenum target, GLuint textureID );`
where \( \text{target} \) is \( \text{GL_TEXTURE_1D}, \text{GL_TEXTURE_2D}, \) or \( \text{GL_TEXTURE_3D} \).
- Tells OpenGL which "texture id" we will be working with currently.
- If \( \text{textureID} \) is being used for the first time a new texture object is created and assigned the ID = \( \text{textureID} \). This is now the active texture.
- If \( \text{textureID} \) has been used before, the texture object with ID = \( \text{textureID} \) becomes the active texture.

Specifying a 2-d Texture Object

OpenGL provides a flexible way to define a texture, but there are many parameter values to be specified:

- **External Format**: The format in which you present your texture to OpenGL (e.g., RGB, RGBA, ...)
- **Internal Format**: The format in which OpenGL stores the texture internally and the number of bits per pixel.
- **Width, height**: Size of the image. For technical reasons, OpenGL requires image sizes to be a power of 2, but you can always pad your images out with extra pixels to satisfy this requirement.
- **Level of detail**: It is possible to provide images in different levels of detail. We will discuss this when we talk about MIP-mapping.
- **Image border**: For purposes of smoothing, OpenGL interpolates a pixel’s colors with its neighbors. This is a problem along the image edge, since there are no neighbors. Thus, a thin border around the image can be provided for smoothing purposes.
Specify a 2-d Texture Object

```c
void glTexImage2D ( GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid* texels );
```

**Sample:**
```c
glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGBA, 128, 128, 0, GL_RGBA, GL_UNSIGNED_BYTE, myImage );
```
- (format) and (type) are used to specify the way in which the texels are stored in your image array.
- (internalFormat) specifies how OpenGL should store the data internally.
- (width) and (height) give the image size. They must be powers of 2. You can use `gluScaleImage` to scale your image.
- (level) and (border) have other uses (see documentation).

Specify how Texture is Applied

How is the color of the texture pixel combined with the existing pixel?

The main one issue to do with whether the texture color is combined with existing object color after lighting (modulation) or is just painted on (decal).

```c
void glTexEnvf ( GLenum target, GLenum pname, TYPE value );
```

where (target) is: `GL_TEXTURE_ENV`.

- (pname) `GL_TEXTURE_ENV_MODE`
- (value) `GL_MODULATE` (mix with lighting) or, `GL_REPLACE` (just paint this color).
Specify how Texture is Applied

There are also parameters that specify how the texture is to be mapped. These involve issues such whether the texture should wrap around (repeat) and how to magnify/shrink it.

\[ \text{glTexParameter}(\text{if}) \ (\text{GLenum} \ \text{target}, \ \text{GLenum} \ \text{pname}, \ \langle \text{TYPE} \rangle \langle \text{value} \rangle); \]

where \( \text{target} \) can be: \( \text{GL\_TEXTURE\_1D}, \text{GL\_TEXTURE\_2D}, \ldots \)

\[ \langle \text{pname} \rangle \langle \text{value} \rangle \]

<table>
<thead>
<tr>
<th>\langle \text{pname} \rangle</th>
<th>\langle \text{value} \rangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{GL_TEXTURE_WRAP_S}</td>
<td>\text{GL_CLAMP}, \text{GL_REPEAT}</td>
</tr>
<tr>
<td>\text{GL_TEXTURE_WRAP_T}</td>
<td>\text{GL_CLAMP}, \text{GL_REPEAT}</td>
</tr>
<tr>
<td>\text{GL_TEXTURE_MAG_FILTER}</td>
<td>\text{GL_NEAREST}, \text{GL_LINEAR}, \ldots</td>
</tr>
<tr>
<td>\text{GL_TEXTURE_MIN_FILTER}</td>
<td>\text{GL_NEAREST}, \text{GL_LINEAR}, \ldots</td>
</tr>
</tbody>
</table>

Beware: OpenGL's default value for \text{GL\_TEXTURE\_MIN\_FILTER} is very strange. Always specify a value for this parameter.

 Enable the Texture and Draw

\[ \text{glEnable} \ (\text{GL\_TEXTURE\_2D}); \]

- Enable 2-d texturing.

\[ \text{glTexCoord2f}(\text{GL\_FLOAT} \ s, \ \text{GL\_FLOAT} \ t); \]

- Specify texture coordinates for the next vertex. (Applies to all subsequent vertices until changed, just the same as \text{glNormal}(), and \text{glColor}().)
- Indexing is relative to the lower left corner, and (irrespective of the image size), \( s \) and \( t \) range from 0 to 1.

\[ \text{glDisable}(\text{GL\_TEXTURE\_2D}); \]

- Disable 2-d texturing, to return to simple coloring.
Example

Texture Initialization:
```c
glGenTextures (...); // create new texture objects
glBindTexture (...); // make this texture active
glTexParameteri (...); // define texture properties
// ... input texture array from file or generate ...
glTexImage2D (...); // provide the texture to OpenGL
```

Displaying a Textured Object:
```c
glEnable (GL_TEXTURE_2D); // enable texturing
glBindTexture (...); // activate the desired texture
glBegin (GL_TRIANGLES); // draw the object
glTexCoord2f (...); glNormal3f (...); glVertex3f (...);
// ... (draw other vertices in the same way)
glEnd ();
glDisable (GL_TEXTURE_2D); // done
```

In-Class Exercise

Skybox:
- You want to enclose your world within a huge cube (skybox), which will be texture mapped with a picture of the sky.
- The skybox is centered about the origin. It is 50,000 units high and 100,000 units wide.
- You want to paste the 1024 x 1024 texture shown below onto the top and sides of your skybox.
- Give the OpenGL commands to draw one of the sides (say the East side) and provide the texture coordinates. Draw the side so that (when viewed from inside) the vertices are given in CCW order.
Minimization Filtering and MIP-mapping

What if one screen-space pixel overlaps many texture pixels?
Ideally we should average these pixels, but this takes time. So OpenGL just takes one.
Result: A jagged appearance, aliasing.
MIP-mapping: Precompute averages and build hierarchy based on powers of 2.

To render: find the appropriate level in the MIP-map, and use this pixel. This smooths out the jagged lines.

Magnification Filtering

What if many screen-space pixels overlaps one texture pixel?
- This results in a blocky appearance.
- Can we do anything to smooth things out?
Magnification Filtering

Bilinear Interpolation:

- Find **four nearest** texels to fragment center \( \{a, b, c, d\} \)
- **Interpolate** between \( a \) and \( b \) in \( s \) (\( p_0 \)):
  \[
p_0 = (1-s) \cdot a + s \cdot b.
  \]
- **Interpolate** between \( c \) and \( d \) in \( s \) (\( p_1 \)):
  \[
p_1 = (1-s) \cdot c + s \cdot d.
  \]
- **Interpolate** between \( p_0 \) and \( p_1 \) in \( t \) (\( p' \)):
  \[
p' = (1-t) \cdot p_0 + t \cdot p_1.
  \]

Perspective Foreshortening

**Linear Interpolation**: is an **affine** operation, but **perspective** is not.

Thus, linear interpolation **does not** account for perspective foreshortening.

**Result**: A line's midpoint in object space is **not** mapped to midpoint in screen space.

Chapter 6, Slide 27
Copyright © D. M. Mount and A. Varshney

Chapter 6, Slide 28
Copyright © D. M. Mount and A. Varshney
Solution: Perspective-Correct Interpolation

**Perspective-Correct Interpolation:**
Let \( p_0 \) and \( p_1 \) be two points to interpolate, and let \( z_0 \) and \( z_1 \) denote their respective depth values.
Let \( q_0 = p_0 / z_0 \) and \( q_1 = p_1 / z_1 \).

**The Trick:** Interpolate \( q_0 \) and \( q_1 \) and then divide by the interpolant of \( 1/z_0 \) and \( 1/z_1 \). Effectively this is the same as doing linear interpolation prior to perspective normalization, and then projecting. Letting \( 0 \leq \alpha \leq 1 \) be the interpolation parameter, we have:

\[
q_\alpha = \frac{(1-\alpha)q_0 + \alpha q_1}{(1-\alpha)/z_0 + \alpha/z_1}
\]

\[
p_\alpha = \frac{(1-\alpha)p_0 + \alpha p_1}{(1-\alpha)/z_0 + \alpha/z_1}
\]

---

**Overview**

- The texture-mapping process
- Texture mapping in OpenGL
  - Other types of maps: Bump, Displacement, Environment
- Procedural textures
- Perlin noise
Bump Mapping

Bump Mapping:
- Introduced by Blinn in 1978.
- We perceive bumps and surface roughness because of variations in diffuse and specular illumination.
- In turn, these are functions of the surface normal vector: \((n \cdot l)\) and \((n \cdot h)^\alpha\).
- Remarkably by modifying just the surface normal \(n\), we can create appearance of bumps without actually changing the surface geometry.
- Note: These are not "real" bumps. Silhouettes remain smooth.

Image courtesy SIGGRAPH Education Materials

Chapter 6, Slide 31
Copyright © D. M. Mount and A. Varshney

Bump Mapping

Images courtesy Michael Capps (NPS)

Chapter 6, Slide 32
Copyright © D. M. Mount and A. Varshney
Displacement Mapping

Displacement Mapping:
- In contrast to bump mapping, the actual geometry is displaced.
- The surface as well as silhouettes appear non-smooth.
- Presents a challenge for deferred shading systems (when shading is performed after visibility).

Environment Mapping

Environment Mapping:
- Invented by Blinn and Newell 1976.
- Index into texture map not by (u, v) but by reflection rays.
- Creates illusion of reflectivity.
Environment Mapping

Environment Mapping:
- The methods we have discussed so far map the texture to a fixed location on the surface.
- Environment mapping determines texture coordinates in a view-dependent manner.

Computing Reflection Color (in the ideal case):
- Let \( p \) be a point on the object's surface. Let \( v \) be the view vector (towards the viewer) and let \( n \) be the surface normal vector.
- Let \( r \) be the reflection vector (a function of \( v \) and \( n \)). The color at point \( p \) is the color of the environment in the direction \( r \).

Questions:
- How do we compute \( r \)?
- How do we compute the environment color in the direction \( r \)?

Computing the Reflection Vector:
- Given \( v \) and \( n \) compute \( r \).
- Assume \( v \) and \( n \) normalized.
- From properties of the dot product, recall that the orthogonal projection of \( v \) onto \( n \) is:
  \[
  n' = ((v \cdot n) / (n \cdot n)) n = (v \cdot n) n.
  \]
- The vector from the head of \( v \) to head of \( n' \) is \( u = n' - v \).
- The reflection vector is
  \[
  r = v + 2u = v + 2(n' - v) = 2(v \cdot n) n - v.
  \]
Environment Mapping

Computing the Color in Direction $r$:
- If we had a full 3-d environment model, we could shoot a ray starting at $p$ in the direction $r$ and determine what is hit.
- But this too computationally expensive for interactive speeds.
- Instead, we pre-compute a panoramic image of the environment from a point centered inside the object. Can be stored as 6 images projected onto the faces of a cube.

![Diagram of environment mapping](image1)

Chapter 6, Slide 37
Copyright © D. M. Mount and A. Varshney

Environment Mapping

Computing the Color in Direction $r$:
- Given the reflection vector $r$, we compute the point of the cube would hit by the ray $p + r$, and use this color. (See Fig 1.)
- To further simplify the computation, we can ignore $p$ and just shoot the vector from the center of the box. (See Fig 2.)
- This maps all parallel reflection vectors to the same point, but it is very fast and looks okay if the environment is far from the object.

![Diagram of environment mapping](image2)

Chapter 6, Slide 38
Copyright © D. M. Mount and A. Varshney
Environment Mapping

How to compute \( s \) and \( t \)?

- Let \( r = (a, b, c) \). The component of maximum absolute value \( \max(|a|, |b|, |c|) \) determines which face of the cube \( r \) hits.
- Suppose that \( |a| \) is maximum and positive, that is, the top of the cube. (There are 5 other analogous cases.)
- Texture coordinates \( s \) and \( t \) are determined by the slopes, \( (b/a) \) and \( (c/a) \). These slopes are in the interval \([-1,+1]\), and so we transform to map to the range \([0,1]\), yielding:

\[
\begin{align*}
    s &= \frac{1}{2} \left( \frac{b}{a} + 1 \right) \\
    t &= \frac{1}{2} \left( \frac{c}{a} + 1 \right)
\end{align*}
\]

Overview

- The texture-mapping process
- Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
  - Procedural textures
- Perlin noise
Procedural Textures

Procedural Textures:
Textures need not come from images. A procedural texture is generated by a function \( f(s, t) \), which generates a color for each texture coordinate.

Example: Checkerboard texture.

Making the Checkerboard:
Consider with a 1-d vertical checkerboard pattern with 8 columns. The column subdivisions are at multiples of \( \frac{1}{8} \).

So we have:
- Column 0: \( 0 \leq 8 \cdot s < 1 \) (white)
- Column 1: \( 1 \leq 8 \cdot s < 2 \) (black)
- Column 2: \( 2 \leq 8 \cdot s < 3 \) (white)
- Column 3: \( 3 \leq 8 \cdot s < 4 \) (black)

Final 1-d Rule:
- \( [8 \cdot s] \mod 2 = 0 \rightarrow \text{white} \)
- \( [8 \cdot s] \mod 2 = 1 \rightarrow \text{black} \)
**Procedural Textures**

**Making the Checkerboard:**
Consider with a 1-d horizontal checkerboard pattern with 8 columns.

By analogy we have:
- \([8 \cdot t] \mod 2 = 0 \rightarrow \text{white}\)
- \([8 \cdot t] \mod 2 = 1 \rightarrow \text{black}\)

**Final 2-d Checkboard Rule:**
To combine these we see that a square is **white** if and only if both the s-rule and t-rule generate the same color, and black otherwise. We have the following final rule (where ⊕ denotes exclusive-or):

\[
f(s, t) = \text{white} \text{ if } ([8 \cdot s] \mod 2 \oplus [8 \cdot t] \mod 2) = 0 \text{ and black otherwise.}
\]

Of course this is easy to generalize to values other than 8.

---

**Examples of Procedural Models**

Images from Texturing and Modeling: A Procedural Approach
By Ebert, Musgrave, Peachey, Perlin, and Worley

Chapter 6, Slide 44
Copyright © D. M. Mount and A. Varshney
Overview

- The texture-mapping process
- Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
- Procedural textures
  - Perlin noise
Perlin Noise

**Natural phenomena:** derive their richness from random variations.
- But we don't want white noise!
- Perlin noise provides nicely "structured" noise.
- Developed by Ken Perlin in 1980's.

**Properties:**
- Reproducible. (Generated by a function so there is no need to store large texture images.)
- No repetitions.
- Band limited (smoothly changing).
- User-specifiable dynamic range.

---

Perlin Noise

**Perlin noise:** Is somewhat like like a fractal.
The overall noise function is a layering of many simpler noise functions. These functions are structurally similar, but vary in frequency and amplitude:

**Terms from Harmonic Analysis:**
- **Wavelength:** The distance between successive crests.
- **Frequency:** The number of crests per unit distance, \(1/\text{wavelength}\).
- **Amplitude:** Height of the crests.
Harmonics

**Periodic Functions**: can be generated with a given frequency $\omega$ and amplitude $a$ by an appropriate transformation of the sine function:

$$\sin t \rightarrow f_{c,\omega}(t) = a \cdot \sin(\omega \cdot t)$$

**Periodic Noise**: Suppose that we are given a pseudo-random noise function that is periodic in nature. (See below.) We can transform its frequency and amplitude in the same manner:

$$\text{noise}(t) \rightarrow \text{noise}_{c,\omega}(t) = a \cdot \text{noise}(\omega \cdot t)$$

Perlin: Layering and Dampening

**How to Layer?** It is common to layer functions by increasing frequency by a doubling, and decreasing amplitude by halving:

$$\text{noise}(t) = \sum_{i=0}^{\infty} \frac{1}{2^i} \cdot \text{noise}(2^i \cdot t)$$

**Dampening**: The above is rather limited. We can generalize this by providing a parameter called dampening factor:

- frequency$(i) = 2^i$
- amplitude$(i) = 1/d^i$, where $d$ is the dampening factor. Thus we have:

$$\text{noise}_p(t) = \sum_{i=1}^{\infty} \frac{1}{d^i} \cdot \text{noise}(2^i \cdot t)$$

As the dampening factor increases, the noise smooths out.
Perlin: Octaves

Octaves:
- Each successive layer has **twice the frequency** as the previous one.
- The individual noise layers are called **octaves**.
- Why is it called an **octave**?
  - In music an octave is the span between 8 whole notes.
  - **Doubling** the frequency increases the pitch by one octave.

How many octaves should I use?
- Up to you. The **more octaves** the higher the **final frequency**.
- **No need to generate frequencies** higher than the **pixel sampling rate** on your image, since such tinier variations will not be visible.

Perlin: Basic Noise Function

All this assumes that we have a **basic noise function** noise(x).

How do we generate it?
- **Random**? ...but this would not reproduce the same function each time. (Our texture would vary from frame to frame.)
- **Pseudo-random**: A function that appears random, but produces the same value each time it is invoked with the same argument.
- See [http://freespace.virgin.net/hugo.elias/models/m_perlin.htm](http://freespace.virgin.net/hugo.elias/models/m_perlin.htm) for a sample pseudo-random noise function that maps a 32-bit integer to a pseudo-random float in the interval [-1,+1].

```cpp
float noise (int x) {
    x ^= (x << 13)^x;
    return ( 1.0 - ( (x * x * x * 15731 + 789221) + 1376312589) & 7fffffff) / 1073741824.0;
}
```

If you do not understand this, don't worry!
Perlin: Smoothing the Basic Noise

Smoothing:
- To avoid a blocky look, we should smooth the noise out, by interpolating between successive values.
- **Linear interpolation** (or Lerp) is fast, but produces a rather jagged results:
  - Lerp(a, b, x) ← (1-x) · a + x · b, where 0 ≤ x ≤ 1.
- **Cubic interpolation**: is very smooth, but a bit complicated.
- **Cosine interpolation**: is a simple compromise. Idea:
  - The interpolant is (1-f(x)) · a + f(x) · b, where 0 ≤ x ≤ 1.
  - To make the transitions gradual, select a function f(x) that it covers the same interval [0,1], but its derivative is zero at x = 0 and x = 1.
  - f(x) ← (1 – cos (π · x))/2 does the trick.

1-d Perlin Noise: Octaves and Sum

Various octaves of 1-dimensional Perlin noise and their sum.
**Perlin: Generalization to 2-dimensions**

2-d **Noise**: Can be done by making the noise a function of x and y:
- \( \text{noise2D}(x, y) \leftarrow \text{noise}(x + P \cdot y) \),
  where \( P \) is a large prime number. (Watch out for arithmetic overflow.)

2-d **Interpolation**: a generalization of bilinear interpolation:
- Interpolate along x:
  \( p_0 \leftarrow \text{Interp}(a, b, x) \)
  \( p_1 \leftarrow \text{Interp}(c, d, x) \)
- and then interpolate along y:
  \( p' \leftarrow \text{Interp}(p_0, p_1, y) \)

---

**2-d Perlin Noise: Octaves and Sum**

Images courtesy of Paul Bourke

Chapter 6, Slide: 55
Copyright © D. M. Mount and A. Varshney

Chapter 6, Slide: 56
Copyright © D. M. Mount and A. Varshney
**Perlin: Further Smoothing**

**Further Smoothing:**
- Prior to interpolation, we can further smooth the noise out by **averaging** neighboring noise values together.

**1-d smoothing:**
- \( \text{smooth} \_\text{noise}(x) \leftarrow \text{noise}(x)/2 + \text{noise}(x-1)/4 + \text{noise}(x+1)/4 \)

![Diagram of 1-d smoothing](image)

**2-d smoothing:** Apply this same idea to \( x \), and then \( y \).

![Diagram of 2-d smoothing](image)

---

**Perlin: Putting it Together**

**1-d Perlin Noise Function:** (The 2-d case is an exercise.)

```c
float noise (int x) { ... given earlier ... }
float smooth_noise (int x) { ... given earlier ... }
float interp_noise (float x) {
    i ← \text{int} (x)
    frac ← x - i
    return interp ( smooth_noise(i), smooth_noise(i+1), frac )
}
float perlin_noise (float x) {
    d ← dampening\_factor
    n ← number\_of\_octaves
    total ← 0
    for ( 0 ≤ i ≤ n-1 )
        total ← total + (1/d^i) \cdot interp\_noise ( 2^i \cdot x )
    return total
}
```
Summary

Summary:
- Texture mapping
- Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
- Procedural textures
- Perlin noise

What’s Next?
- Shadows and Reflection