Week 11

Particle Systems

2019-03-29

Objectives

- Explore texture atlases, mapping and vertex arrays
- Implement particle systems to create effects
- Implement animation to show an object in motion
- Render textures and utilize shaders to create a distinct look

Texture Atlases

- A **texture atlas** is a single texture that contains multiple objects
  - Also known as a sprite sheet or tile set
  - Results in fewer image files thus reducing switching between textures
  - SFML sprite class `sf::Sprite` contains a texture rectangle containing pixel coordinates
  - Rectangle is of type `sf::IntRect`

Texture Atlases (cont'd.)

- Aircraft, projectile and pickup textures will be merged to one texture, with an ID of `Entities`
- Eventually, we only have the following identifiers:

```
namespace Textures
{
    enum ID
    {
        Entities,
        Jungle,
        TitleScreen,
        Buttons,
        Explosion,
        Particle,
        FinishLine,
    };
}
```

Texture Atlases (cont'd.)

- We modify our data tables to store a texture rectangle in addition to the texture ID as below:

```
struct AircraftData
{
    Textures::ID texture;
    sf::IntRect textureRect;
    ...
};

std::vector<AircraftData> initializeAircraftData()
{
    std::vector<AircraftData> data(Aircraft::TypeCount);
    data[Aircraft::Eagle].texture = Textures::Entities;
    data[Aircraft::Eagle].textureRect = sf::IntRect(0, 0, 48, 64);
    ...

    return data;
}
```

Texture Atlases (cont'd.)

- Then we initialize the sprite with both texture and texture rect:

```
namespace
{
    const std::vector<AircraftData> Table = initializeAircraftData();
}

Aircraft::Aircraft(Type type, const TextureHolder& textures,
    const FontHolder& fonts) : mSprite(
    textures.get(Table[type].texture), // sf::Texture
    Table[type].textureRect) // sf::IntRect
    , ...
{
    centerOrigin(mSprite);
    ...
}
```

Rendering in SFML

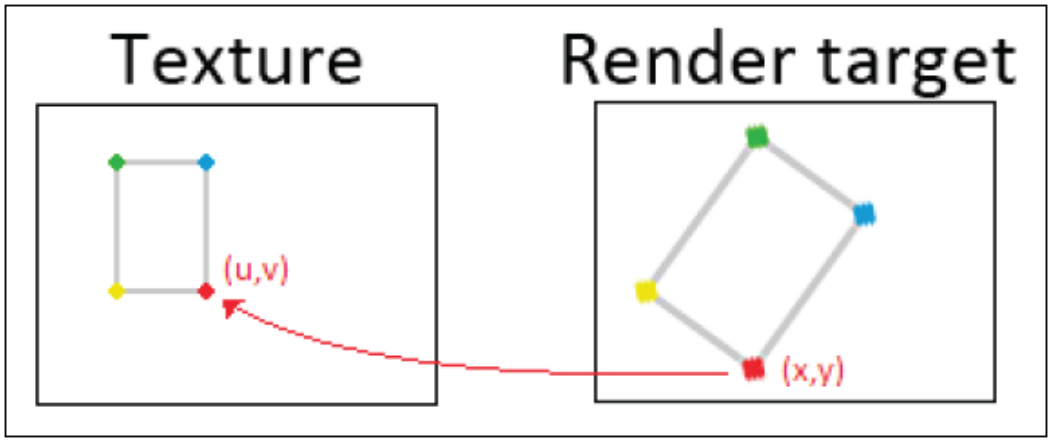
- SFML is built atop the Open Graphics Library (OpenGL)
- A render target is the place where 2D objects such as sprites, text or shapes are rendered
- SFML uses the abstract base class sf::RenderTarget
  - `clear()` and `draw()` methods
- A render window is a concrete implementation of a render target
  - The class is `sf::RenderWindow`

Rendering in SFML (cont'd.)

- A render texture is another realization of the render target concept
  - You do not draw objects to a window
  - Instead to a texture
  - Class is `sf::RenderTexture`

Texture Mapping

- A texel (texture element) is the term used for pixels in texture space
- A vertex is a point that defines the geometry of an object
  - Create lines, triangles, rectangles, etc.
- Texture mapping defines how target coordinates are mapped onto texture coordinates



Texture Mapping (cont'd.)

- SFML provides `sf:Vertex` that represents a vertex
  - `sf::Vector2f position` – the target coordinates (x,y)
  - `sf::Vector2f texCoords` – the texture coordinates (u,v)
  - `sf::Color color` – used to colorize the vertex

Vertex Arrays

- A vertex array is a collection of vertices that are drawn together
  - The class is `sf::VertexArray`
  - Below is a small, incomplete example of how everything interacts:

```
sf::Vertex v;
v.position = sf::Vector2f(x, y);
v.texCoords = sf::Vector2f(u, v);
v.color = sf::Color::Blue;

sf::VertexArray vertices;
vertices.setPrimitiveType(sf::Quads);
vertices.append(v);
...

sf::RenderTarget& target = ...;
target.draw(vertices);
```

Particle Systems

- A particle system manages the behavior of many particles to a desired effect
  - **Emitters** create new particles continuously
  - **Affectors** essentially animate or change particles

- We don`t model each particle to a sprite as each one would have to be drawn separately
- Instead, each particle is modeled as an object with four vertices and then inserted into a single vertex array
- Then, we have a method to draw them all with only one draw call

Particle Systems (cont'd.)

Particle Systems (cont'd.)

```
struct Particle
{
   enum Type
   {
      Propellant,
      Smoke,
      ParticleCount
   };
   sf::Vector2f position;
   sf::Color color;
   sf::Time lifetime;
};

struct ParticleData
{
   sf::Color color;
   sf::Time lifetime;
};
```

Particle Nodes

```
class ParticleNode : public SceneNode
{
public:
    ParticleNode( Particle::Type type, const TextureHolder& textures);
    void addParticle( sf::Vector2f position);
    Particle::Type getParticleType() const;
    virtual unsigned int getCategory() const;
    ...
private:
    std::deque<Particle> mParticles;
    const sf::Texture& mTexture;
    Particle::Type mType;
    mutable sf::VertexArray mVertexArray;
    mutable bool mNeedsVertexUpdate;
};
```

Particle Nodes (cont'd.)

```
void ParticleNode::addParticle(sf::Vector2f position)
{
    Particle particle;
    particle.position = position;
    particle.color = Table[mType].color;
    particle.lifetime = Table[mType].lifetime;
    mParticles.push_back(particle);
}

void ParticleNode::updateCurrent(sf::Time dt, CommandQueue&)
{
    while (!mParticles.empty() && mParticles.front().lifetime <= sf::Time::Zero)
        mParticles.pop_front();

    FOREACH(Particle& particle, mParticles)
        particle.lifetime -= dt;

    mNeedsVertexUpdate = true;
}
```

Particle Nodes (cont'd.)

```
void ParticleNode::drawCurrent(sf::RenderTarget& target,
   sf::RenderStates states) const
{
   if (mNeedsVertexUpdate)
   {
       computeVertices();
       mNeedsVertexUpdate = false;
   }
   states.texture = &mTexture;
   target.draw(mVertexArray, states);
}
```

Particle Nodes (cont'd.)

```
void ParticleNode::computeVertices() const
{
    sf::Vector2f size(mTexture.getSize());
    sf::Vector2f half = size / 2.f;

    mVertexArray.clear();

    FOREACH(const Particle& particle, mParticles)
    {
        sf::Vector2f pos = particle.position;
        sf::Color c = particle.color;
        float ratio = particle.lifetime.asSeconds()
                / Table[mType].lifetime.asSeconds();
        c.a = static_cast<sf::Uint8>(255 * std::max(ratio, 0.f));

        addVertex(pos.x - half.x, pos.y - half.y, 0.f, 0.f, c);
        addVertex(pos.x + half.x, pos.y - half.y, size.x, 0.f, c);
        addVertex(pos.x + half.x, pos.y + half.y, size.x, size.y, c);
        addVertex(pos.x - half.x, pos.y + half.y, 0.f, size.y, c);
    }
}
```

Particle Nodes (cont'd.)

```
void ParticleNode::addVertex(float worldX, float worldY,
   float texCoordX, float texCoordY, const sf::Color& color) const
{
   sf::Vertex vertex;
   vertex.position = sf::Vector2f(worldX, worldY);
   vertex.texCoords = sf::Vector2f(texCoordX, texCoordY);
   vertex.color = color;
   mVertexArray.append(vertex);
}
```

Emitter Nodes

```
class EmitterNode : public SceneNode
{
public:
   explicit EmitterNode(Particle::Type type);
   ...
private:
   sf::Time mAccumulatedTime;
   Particle::Type mType;
   ParticleNode* mParticleSystem;
};
```

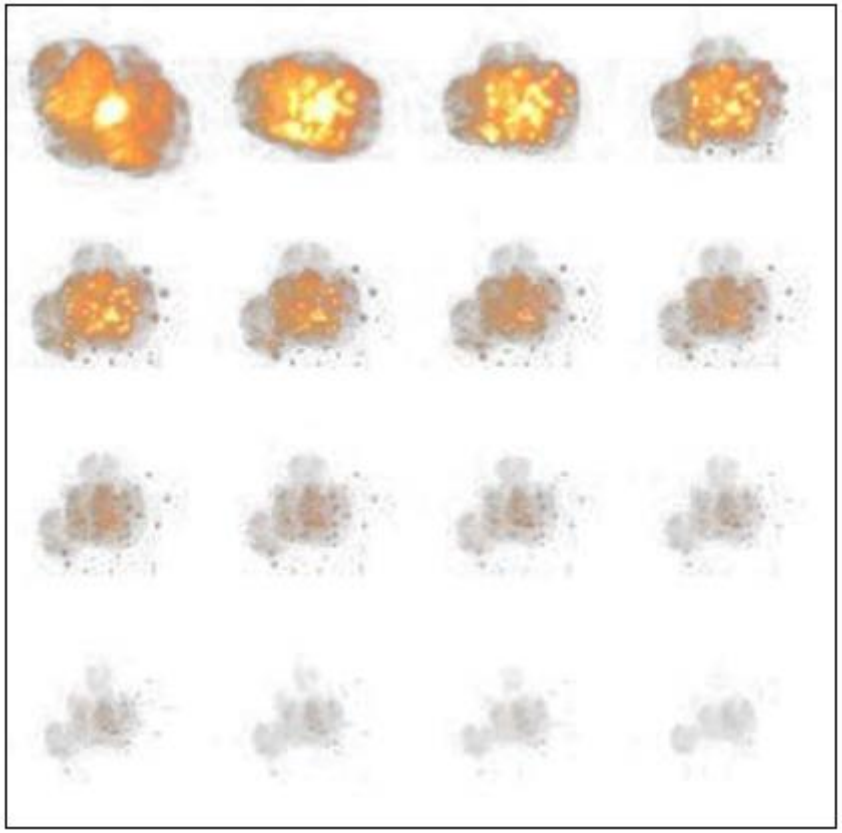
Emitter Nodes (cont'd.)

```
void EmitterNode::updateCurrent(sf::Time dt, CommandQueue& commands)
{
   if (mParticleSystem)
   {
       emitParticles(dt);
   }
   else
   {
       auto finder = [this] (ParticleNode& container, sf::Time)
       {
              if (container.getParticleType() == mType)
                     mParticleSystem = &container;
       };
       Command command;
       command.category = Category::ParticleSystem;
       command.action = derivedAction<ParticleNode>(finder);
       commands.push(command);
   }
}
```

Emitter Nodes (cont'd.)

```
void EmitterNode::emitParticles(sf::Time dt)
{
   const float emissionRate = 30.f;
   const sf::Time interval = sf::seconds(1.f) / emissionRate;
   mAccumulatedTime += dt;
   while (mAccumulatedTime > interval)
   {
       mAccumulatedTime -= interval;
       mParticleSystem->addParticle(getWorldPosition());
   }
}
```

Animated Sprites



Animated Sprites (cont'd.)

```
class Animation : public sf::Drawable, public sf::Transformable
{
public:
   ...
private:
   ...
   sf::Sprite mSprite;
   sf::Vector2i mFrameSize;
   std::size_t mNumFrames;
   std::size_t mCurrentFrame;
   sf::Time mDuration;
   sf::Time mElapsedTime;
   bool mRepeat;
};
```

Animated Sprites (cont'd.)

```
void Animation::update(sf::Time dt)
{
    sf::Time timePerFrame = mDuration / static_cast<float>(mNumFrames);
    mElapsedTime += dt;
    sf::Vector2i textureBounds(mSprite.getTexture()->getSize());
    sf::IntRect textureRect = mSprite.getTextureRect();

    if (mCurrentFrame == 0)
        textureRect = sf::IntRect(0, 0, mFrameSize.x, mFrameSize.y);
```

Animated Sprites (cont'd.)

```
    while (mElapsedTime >= timePerFrame && (mCurrentFrame <= mNumFrames || mRepeat))
    {
       textureRect.left += textureRect.width;
       if (textureRect.left + textureRect.width > textureBounds.x)
       {
         textureRect.left = 0;
         textureRect.top += textureRect.height;
       }
       mElapsedTime -= timePerFrame;
       if (mRepeat)
       {
         mCurrentFrame = (mCurrentFrame + 1) % mNumFrames;
         if (mCurrentFrame == 0)
                   textureRect = sf::IntRect(0, 0, mFrameSize.x, mFrameSize.y);
       }
       else
       {
         mCurrentFrame++;
       }
    }
    mSprite.setTextureRect(textureRect);
}
```

Post Effects

```
class PostEffect
{
public:
   virtual ~PostEffect();
   virtual void apply(const sf::RenderTexture& input,
       sf::RenderTarget& output) = 0;
   static bool isSupported();
protected:
   static void applyShader(const sf::Shader& shader,
       sf::RenderTarget& output);
};
```

Post Effects (cont'd.)

```
void World::draw()
{
   if (PostEffect::isSupported())
   {
      mSceneTexture.clear();
      mSceneTexture.setView(mWorldView);
      mSceneTexture.draw(mSceneGraph);
      mSceneTexture.display();
      mBloomEffect.apply(mSceneTexture, mTarget);
   }
   else
   {
      mTarget.setView(mWorldView);
      mTarget.draw(mSceneGraph);
   }
}
```

Post Effects (cont'd.)

```
void PostEffect::applyShader(const sf::Shader& shader, sf::RenderTarget& output)
{
    sf::Vector2f outputSize = static_cast<sf::Vector2f>(output. getSize());

    sf::VertexArray vertices(sf::TrianglesStrip, 4);
    vertices[0] = sf::Vertex(sf::Vector2f(0, 0), sf::Vector2f(0, 1));
    vertices[1] = sf::Vertex(sf::Vector2f(outputSize.x, 0), sf::Vector2f(1, 1));
    vertices[2] = sf::Vertex(sf::Vector2f(0, outputSize.y), sf::Vector2f(0, 0));
    vertices[3] = sf::Vertex(sf::Vector2f(outputSize), sf::Vector2f(1, 0));

    sf::RenderStates states;
    states.shader = &shader;
    states.blendMode = sf::BlendNone;
    output.draw(vertices, states);
}
```

Shaders

- A shader is a program that is executes on the data you provide
  - Vertices, textures and more
- Bloom: Bloom is an effect which tries to mimic a defect in our eyes and cameras. The effect has been a bit exaggerated in this demonstration picture, but this is what we are going for. So how will we do this by using shaders?

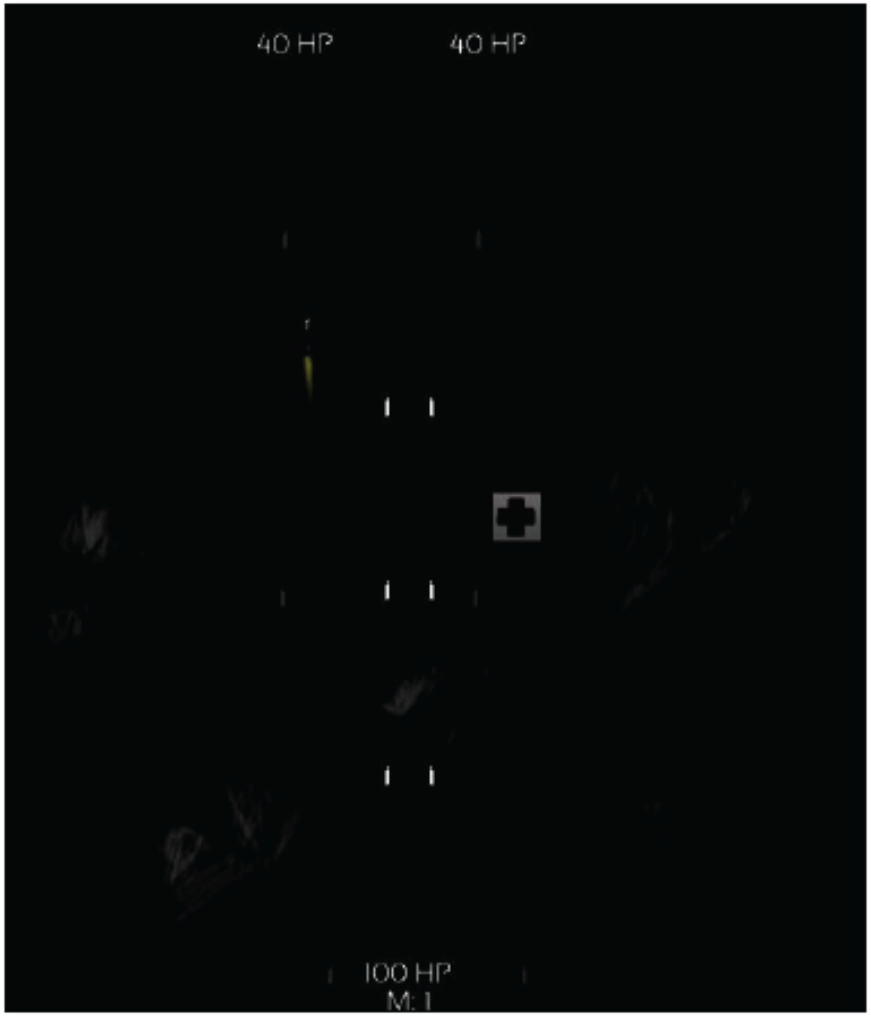
Bloom (Shader Effect)

- Bloom or glow is a computer graphics effect used in video games to reproduce an imaging artifact of real-world cameras.
- The effect produces fringes of light extending from the borders of bright areas in an image, creating an illusion of an extremely bright light.
- It was widely used after the article published by Tron 2.0 authors (Monolith Productions) in 2004.

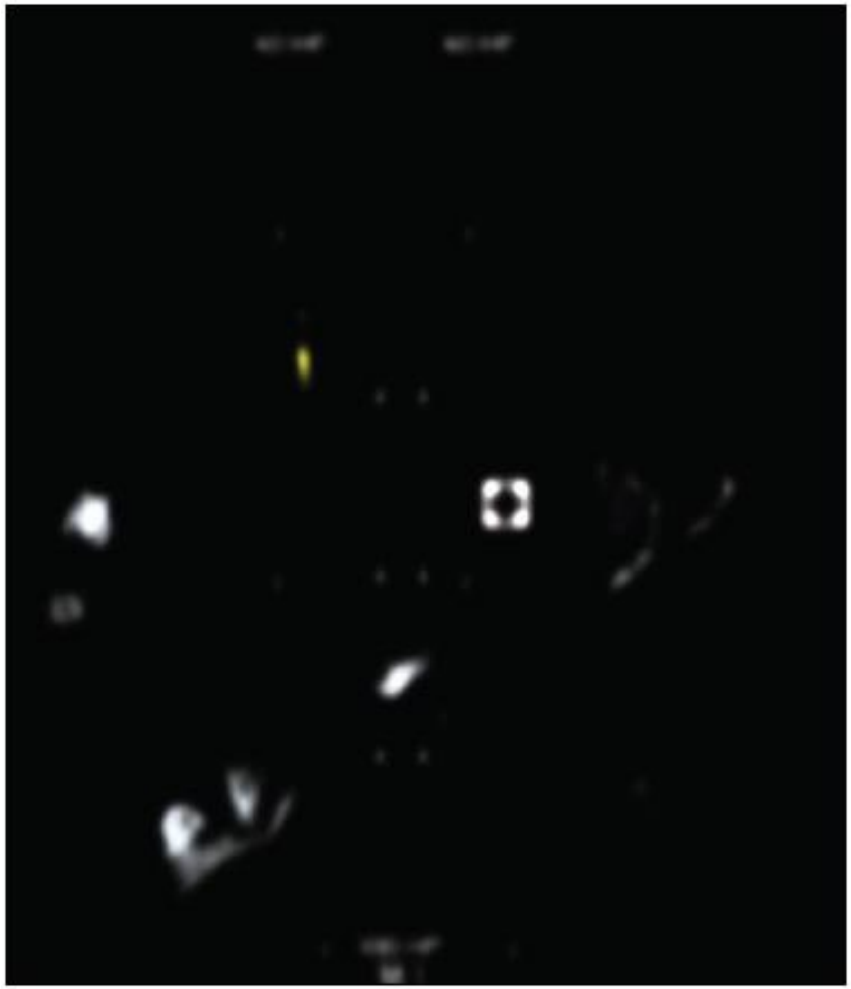
Bloom

- To achieve the bloom effect, we have multiple shader passes. The output of a shader program can only be used as input in the next pass, which limits the possible operations per pass.
- Source image:

- The first shader is the brightness pass; we filter out what is bright and what is not bright in the image by a simple threshold. The resulting image is mostly black:
- Here we can see the different bright colors that should receive the bloom effect.
- Mostly, it is the bullets that we have made white just for this purpose.



- If we simply added these colors to the scene colors, we wouldn't get the bloom effect.
- We would just get a very bright screen. That's not enough. We have to smooth out the texture. This is done by scaling down the texture and performing a Gaussian blur on it. This is done twice, so we end up with a result that is a quarter size of the original texture.

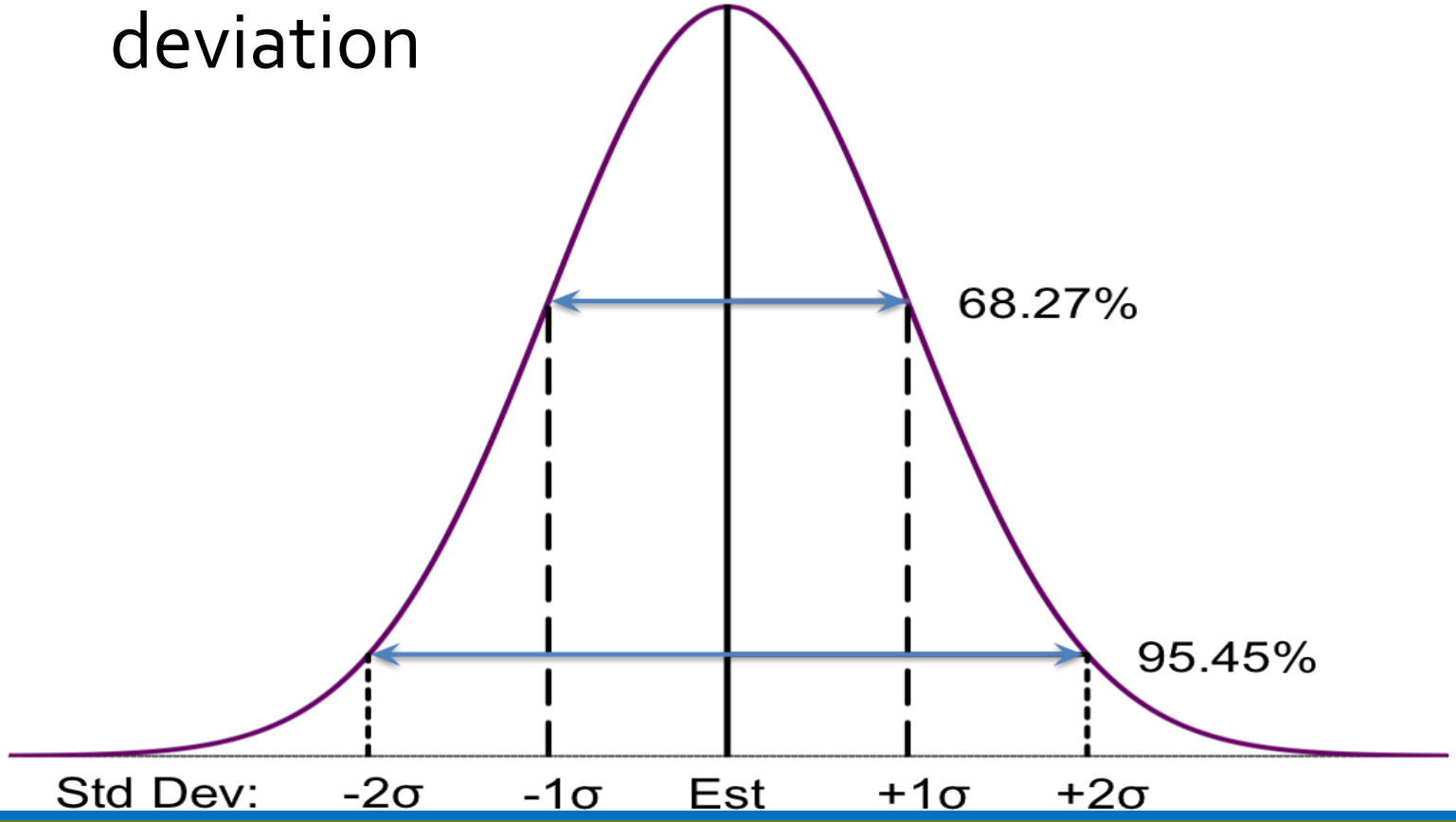


Gaussian

- In probability theory, the normal (or Gaussian) distribution is a very common continuous probability distribution.
- It states that averages of samples of observations of random variables independently drawn from independent distributions converge in distribution to the normal, that is, they become normally distributed when the number of observations is sufficiently large.
- The normal distribution is sometimes informally called the **bell curve**.

Gaussian Distribution

- 2/3rd or 68% percent of Gaussian distribution is within one standard deviation



Gaussian Blur

- In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function (named after mathematician Carl Friedrich Gauss).
- It is a widely used effect in graphics software, typically to reduce image noise and reduce detail.

The difference between a small and large Gaussian blur

Bloom (cont'd.)

- Now it's starting to look like something. For demonstration purposes, we cranked up the effect to make it impossible to miss.

Bloom (cont'd.)

```
void BloomEffect::apply(const sf::RenderTexture& input, sf::RenderTarget& output)
{
    prepareTextures(input.getSize());
    filterBright(input, mBrightnessTexture);

    downsample(mBrightnessTexture, mFirstPassTextures[0]);
    blurMultipass(mFirstPassTextures);

    downsample(mFirstPassTextures[0], mSecondPassTextures[0]);
    blurMultipass(mSecondPassTextures);

    add(mFirstPassTextures[0], mSecondPassTextures[0], mFirstPassTextures[1]);
    mFirstPassTextures[1].display();
    add(input, mFirstPassTextures[1], output);
}
```

Bloom (cont'd.)

```
void BloomEffect::prepareTextures(sf::Vector2u size)
{
   if (mBrightnessTexture.getSize() != size)
   {
      mBrightnessTexture.create(size.x, size.y);
      mBrightnessTexture.setSmooth(true);
      mFirstPassTextures[0].create(size.x / 2, size.y / 2);
      mFirstPassTextures[0].setSmooth(true);
      mFirstPassTextures[1].create(size.x / 2, size.y / 2);
      mFirstPassTextures[1].setSmooth(true);
      mSecondPassTextures[0].create(size.x / 4, size.y / 4);
      mSecondPassTextures[0].setSmooth(true);
      mSecondPassTextures[1].create(size.x / 4, size.y / 4);
      mSecondPassTextures[1].setSmooth(true);
   }
}
```

Bloom (cont'd.)

```
void BloomEffect::filterBright(const sf::RenderTexture& input, sf::RenderTexture& output)
{
    sf::Shader& brightness = mShaders.get(Shaders::BrightnessPass);

    brightness.setParameter("source", input.getTexture());
    applyShader(brightness, output);
    output.display();
}

void BloomEffect::blurMultipass(RenderTextureArray& renderTextures)
{
    sf::Vector2u textureSize = renderTextures[0].getSize();

    for (std::size_t count = 0; count < 2; ++count)
    {
         blur(renderTextures[0], renderTextures[1], sf::Vector2f(0.f, 1.f /
         textureSize.y));
         blur(renderTextures[1], renderTextures[0], sf::Vector2f(1.f /
         textureSize.x, 0.f));
    }
}
```

Bloom (cont'd.)

```
void BloomEffect::blur(const sf::RenderTexture& input, sf::RenderTexture& output, sf::Vector2f
    offsetFactor)
{
    sf::Shader& gaussianBlur = mShaders.get(Shaders::GaussianBlurPass);
    gaussianBlur.setParameter("source", input.getTexture());
    gaussianBlur.setParameter("offsetFactor", offsetFactor);
    applyShader(gaussianBlur, output);
    output.display();
}

void BloomEffect::downsample(const sf::RenderTexture& input, sf::RenderTexture& output)
{
    sf::Shader& downSampler = mShaders.get(Shaders::DownSamplePass);
    downSampler.setParameter("source", input.getTexture());
    downSampler.setParameter("sourceSize", sf::Vector2f(input.
    getSize()));
    applyShader(downSampler, output);
    output.display();
}

void BloomEffect::add(const sf::RenderTexture& source, const sf::RenderTexture& bloom,
    sf::RenderTarget& output)
{
    sf::Shader& adder = mShaders.get(Shaders::AddPass);
    adder.setParameter("source", source.getTexture());
    adder.setParameter("bloom", bloom.getTexture());
    applyShader(adder, output);
}
```

GLSL Tutorial

- http://www.lighthouse3d.com/tutorials/glsl-tutorial