TEXTURE BOMBING

CSC 533 GRADUATE PROJECT

WEIWEI LIU

WHAT IS TEXTURE BOMBING?

- Textures are useful for adding visual detail to geometry, but they don't work as well when extended to cover large areas such as a field of flowers, or many similar objects such as a city full of buildings. Such uses require either a very large amount of texture data or repetition of the same pattern, resulting in an undesirable, regular look.

- **Texture bombing** is a technique that places small decorative elements at irregular interval on the surface of an object to help reduce such pattern artifacts.

IDEA

- Divide the UV space into a regular grid of cells.
- Place an image within each cell at a **random** location to achieve irregular effect.
- The final result is the composite of these images over the background.

HOW TO GET PSEUDO-RANDOM NUMBER IN GLSL

Problem:

No built-in random function(like *rand()* in c++) in GLSL.

Solution:

Use a randomMap(texture with random values) to get random numbers.

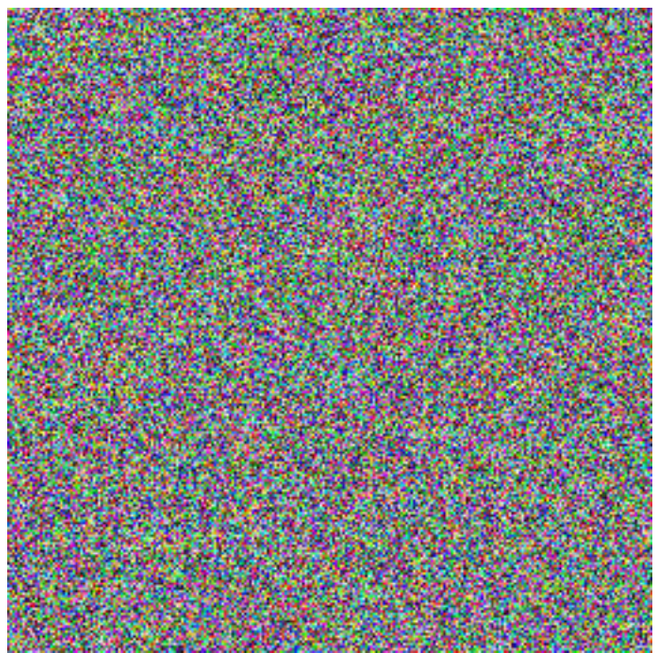


Figure 1. An example of random texture

- The color values for each pixel are obtained by a random number generator.
- When we need a random number, we fetch a vec4 color value from the texture, and use r/g/b component as a random number.
- Ensure each time we access a different position.

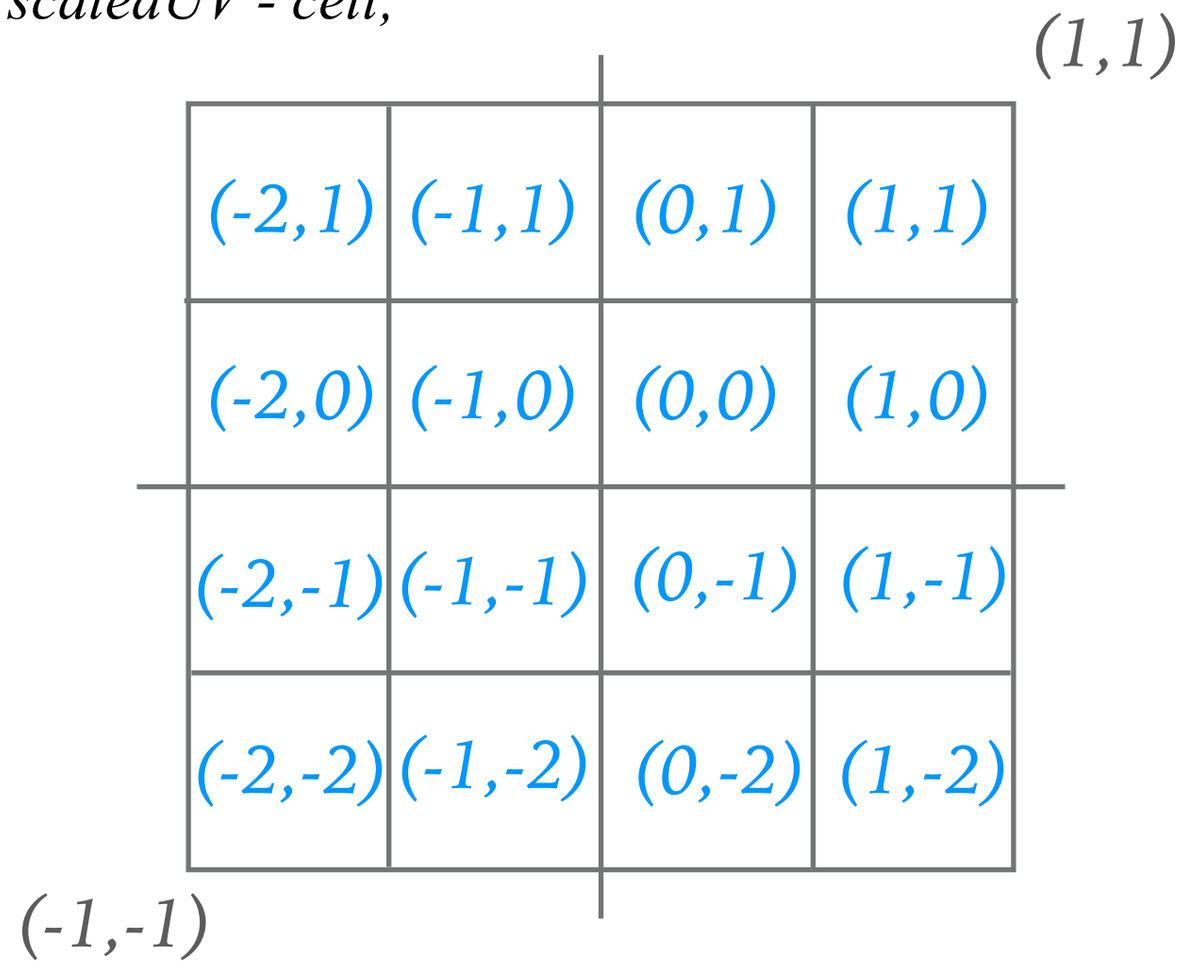
FINDING THE CELL

Compute the cell's coordinates:

*vec2 scaledUV = UV * scale;*

vec2 cell = floor(scaledUV);

float2 offset = scaledUV - cell;



scale = (2, 2)

SAMPLING THE IMAGE

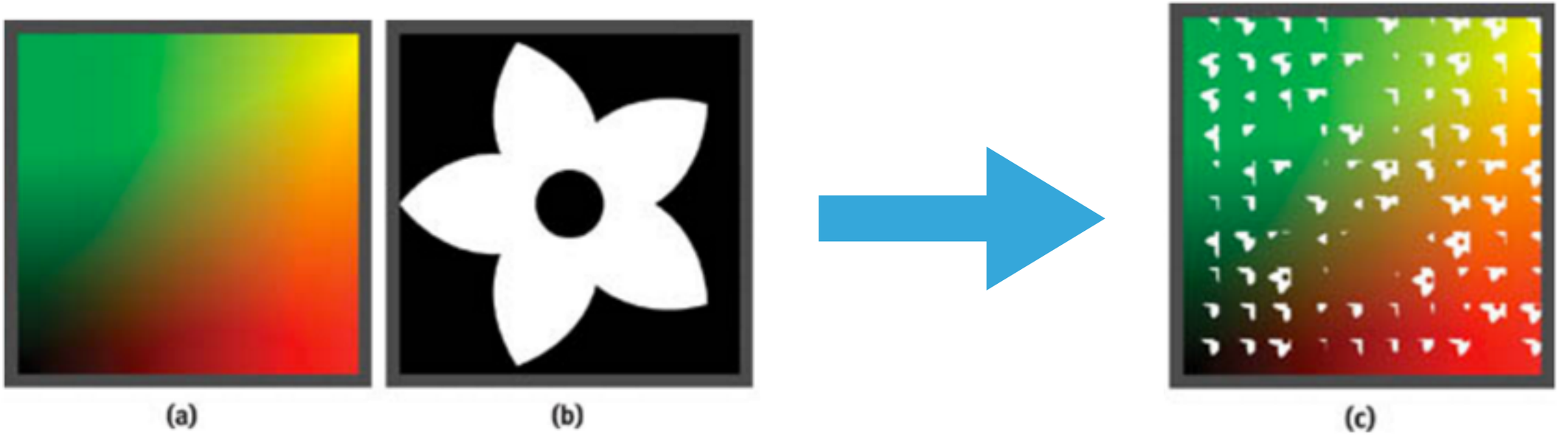
Determine where to put the image.

```
vec2 randomUV = cell.xy * vec2(0.037, 0.119);
vec4 random = texture(randomTex, randomUV);
 // put the image at random.xy in the cell
```

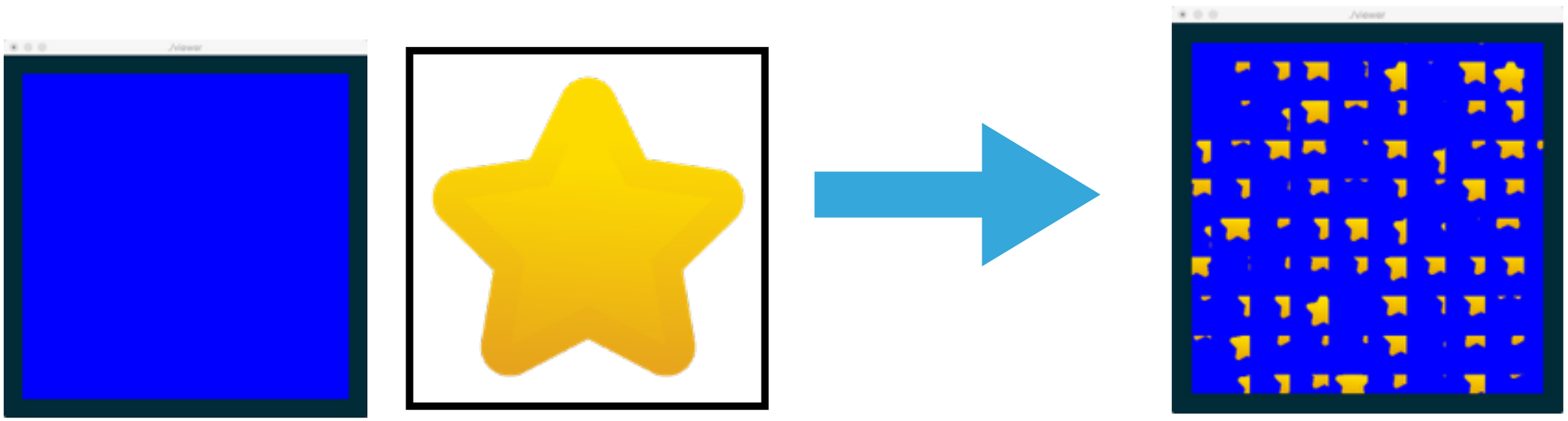
Sample the image:

```
vec4 image = texture(imageTex, offset - random.xy);
if (image.w > 0.0 ) {
    finalcolor.xyz = image.xyz;
}
```

RESULT

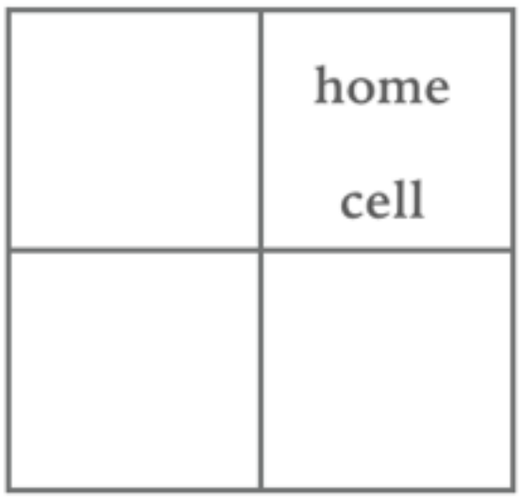


Demo:



ADJACENT CELL'S IMAGE

- Need to check for overlapping images from adjacent cells.
- Since we've limited the image offset relative to a cell to be in (0,1), an image can overlap only those cells above and to the right of its home cell.
- Sample four cells: the current cell plus those cells below and to the left of our cell.



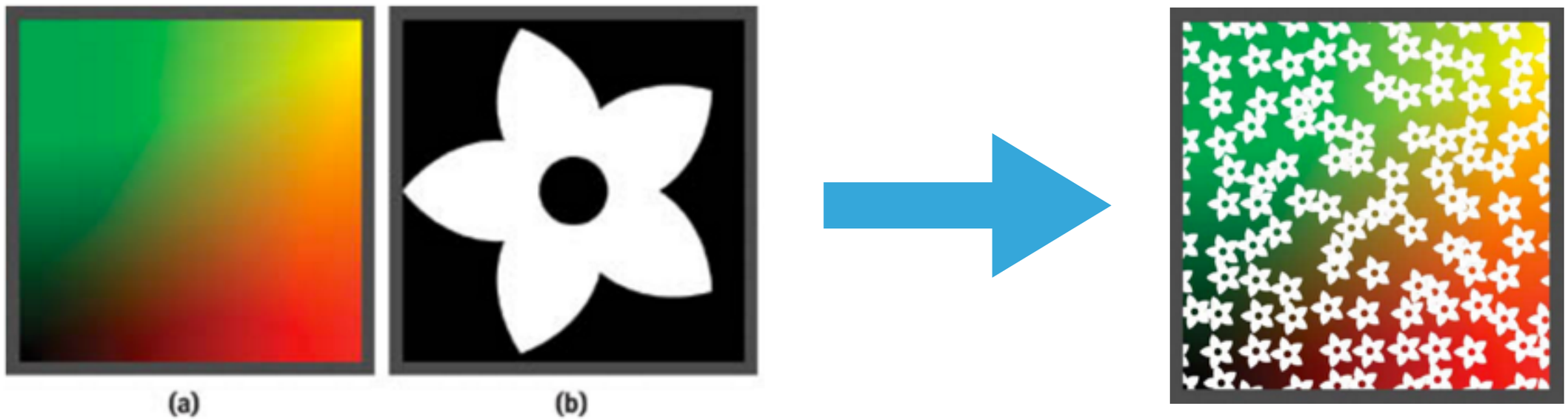
ADJACENT CELL'S IMAGE

```
for (int i = -1; i <= 0; i++) {
    for (int j = -1; j <= 0; j++) {
        vec2 cell_t = cell + vec2(i, j);
        vec2 offset_t = offset - vec2(i, j) ;
        vec2 randomUV = cell_t.xy * vec2(0.037, 0.119);
        vec4 random = texture(randomTex, randomUV);
        vec4 image = texture(imageTex, offset_t - random.xy);
        ……} }
```

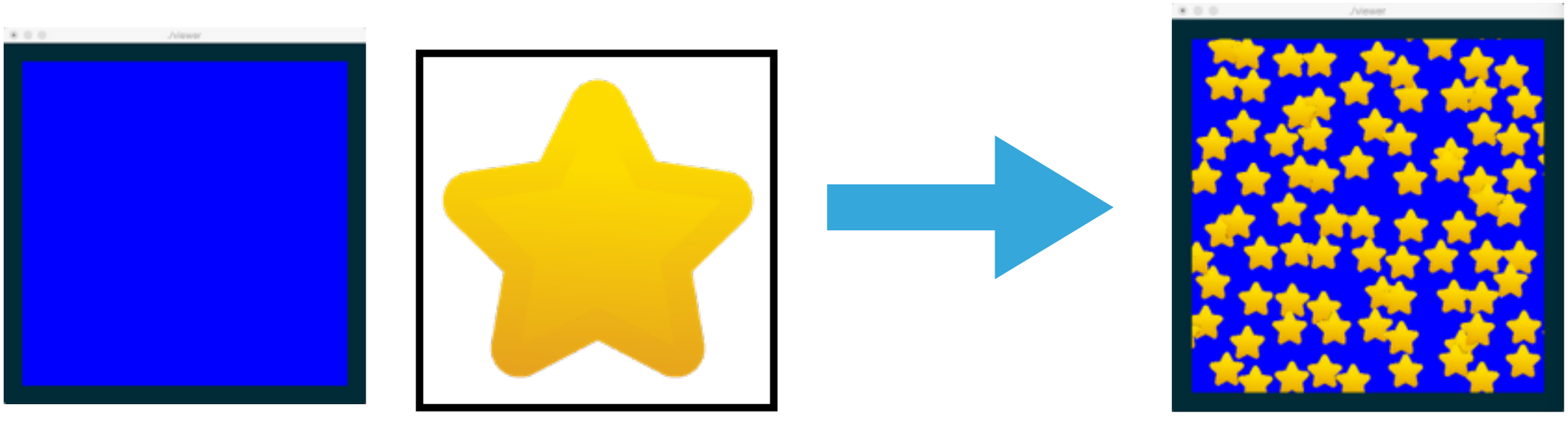
Priority:

- When two images overlap, the one to the upper right is always on top, because it was the last one tested.
- Use random.z to determine the priority.

RESULT



Demo:

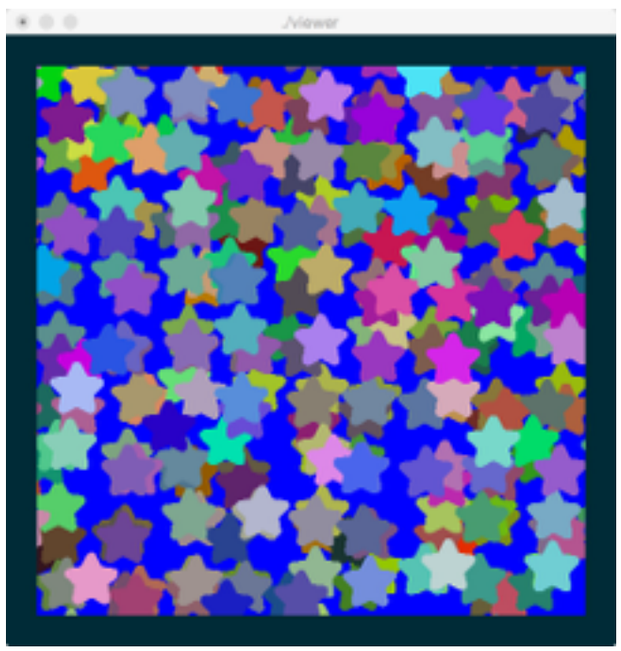


MULTIPLE IMAGES PER CELL

Using only one image per cell can lead to a regular-looking pattern, especially when the cells are small. To counter this tendency, we can increase the number of images in each cell.

```
for (int k = 0; k < NUMBER_OF_SAMPLES; k++) {
    vec4 random = texture(randomTex, randomUV);
    randomUV += vec2(0.03, 0.17);
    ……
}
```

Demo:



PROCEDURAL IMAGES

It's not very efficient to composite images in this manner, because we may have hundreds of images to combine. We may compute the image procedurally.

```
offset_t -= vec2(0.5, 0.5) + vec2(random);
float radius2 = dot(offset_t, offset_t);
if (random.z > priority && radius2 < 0.25) {
    finalcolor.xyz = texture(randomTex, randomUV + vec2(0.13, 0.4)).xyz;
    priority = random.z;
}
```

Demo:

RANDOM IMAGE SELECTION FROM MULTIPLE CHOICES

Another dimension of variation can be added by selecting from one of multiple images for each choice.

Two option:

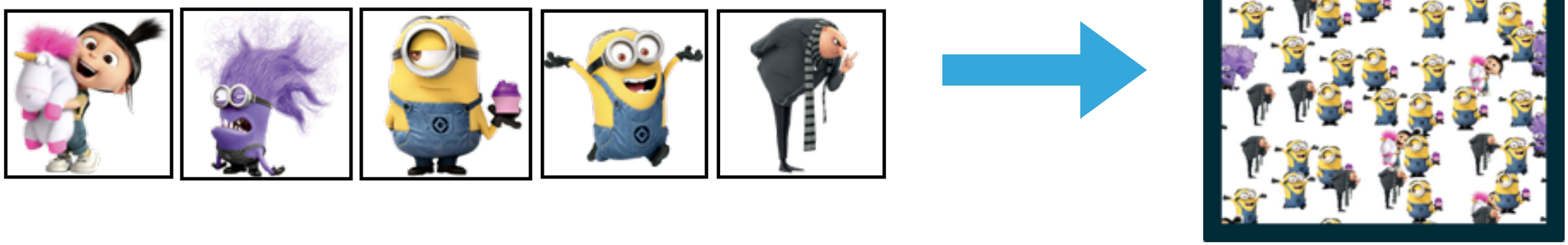
- tile images in a single texture, use random.z as index

- load multiple textures, and use random.z to determine which texture to display.

```
float index = floor(random.x * NUMBER_OF_CHOICES);
vec2 uv = offset_t.xy - random.xy;
vec4 image;
if(index == 0) image = texture(choiceTex1, uv);
else if .....
```

RESULT

Demo:



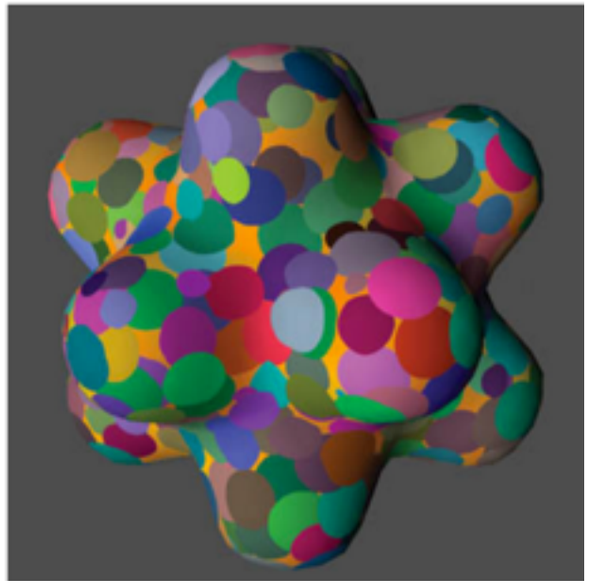
PROCEDURAL 3D BOMBING

We can extend the idea into 3D object.

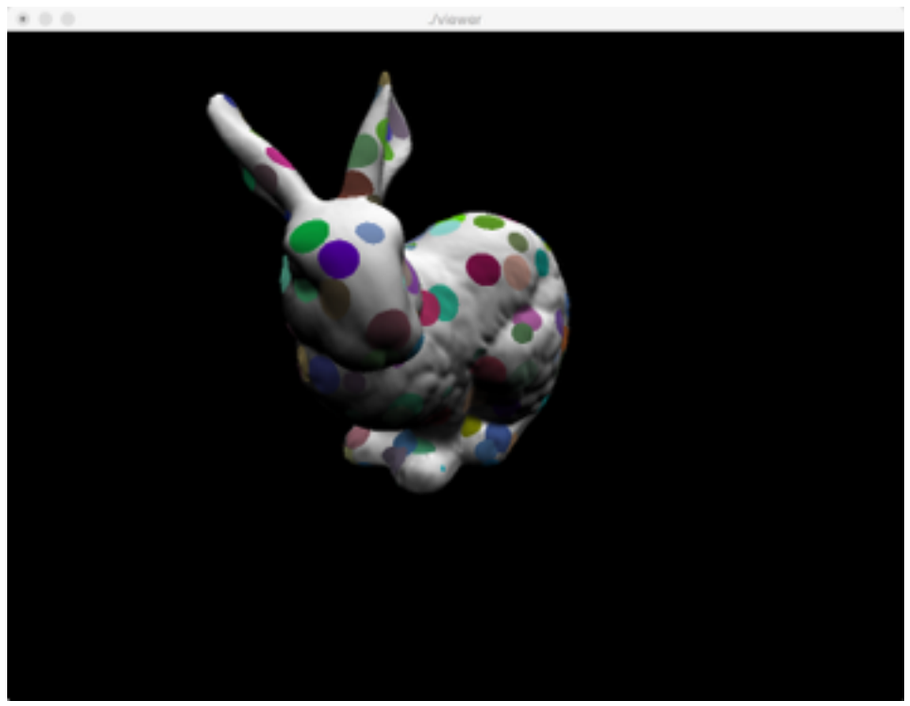
- Divide object space into unit cubes.
- Still use a 2D pseudo-random texture.

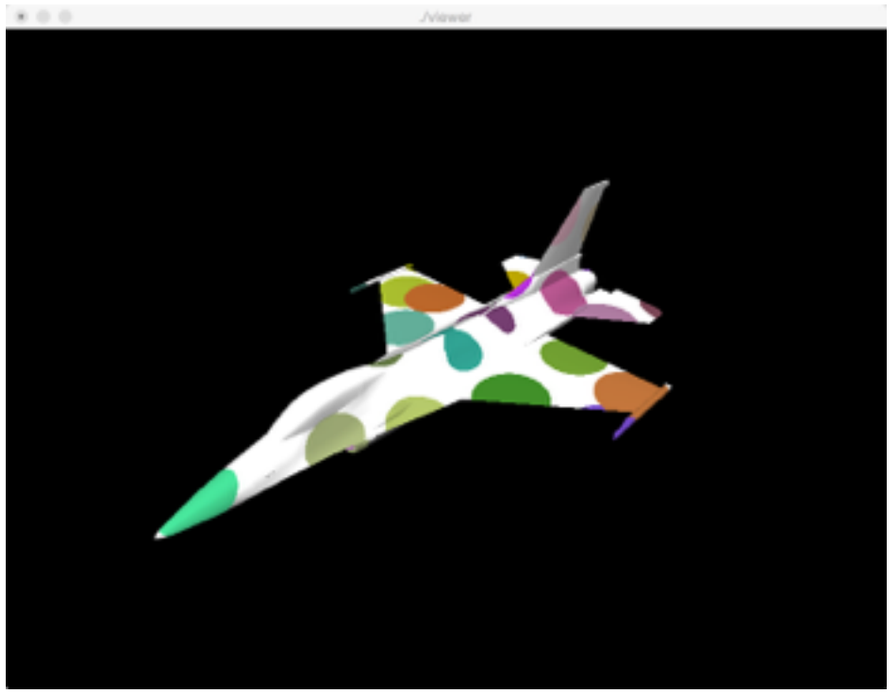
```
for (int i = -1; i <= 0; i++) {
  for (int j = -1; j <= 0; j++) {
    for (int k = -1; k <= 0; k++) {
      vec3 cell_t = cell + vec3(i, j, k);
      vec3 offset_t = offset - vec3(i, j, k);
      vec2 randomUV = cell_t.xy * vec2(0.037, 0.119) + cell_t.z * 0.003;
      ……
```

RESULT

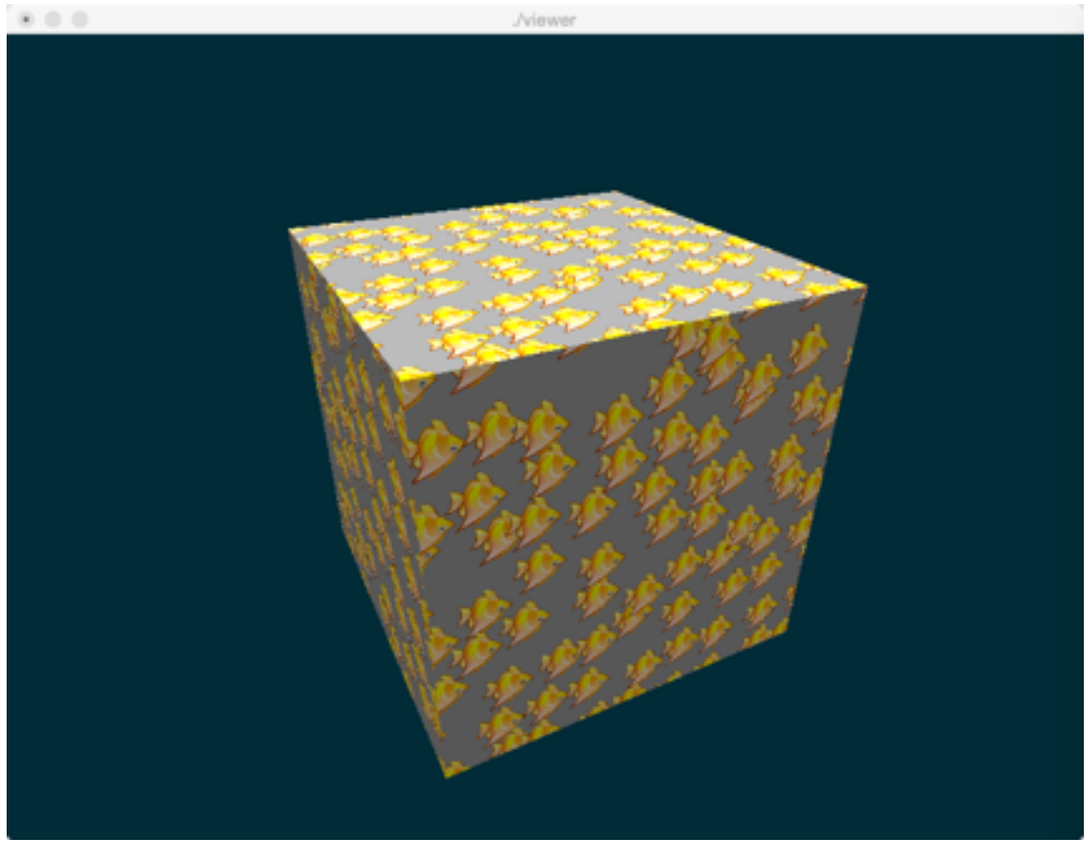


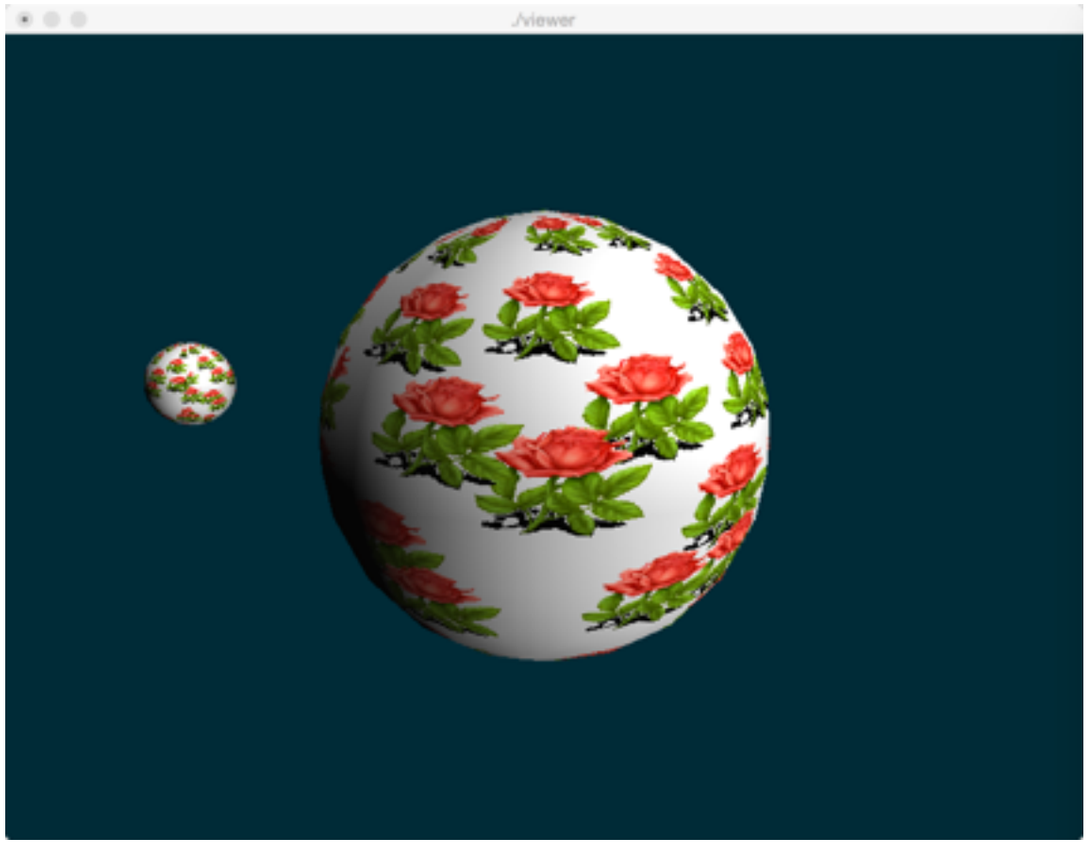
Demo:





3D BOMBING WITH IMAGE FROM TEXTURE





IMPLEMENTATION CONSIDERATIONS

Minification Filter for random texture:

- glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER, GL_LINEAR);

Wrap mode for texture image: (otherwise causing ghost samples)

- glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
- glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);

THANKS!

REFERENCE

Fernando R. GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics[M], Chapter 20. Texture Bombing. Pearson Higher Education, 2004.