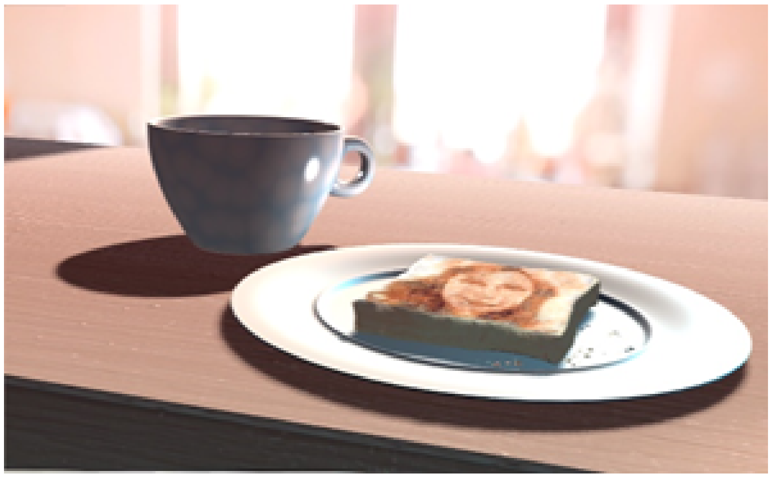
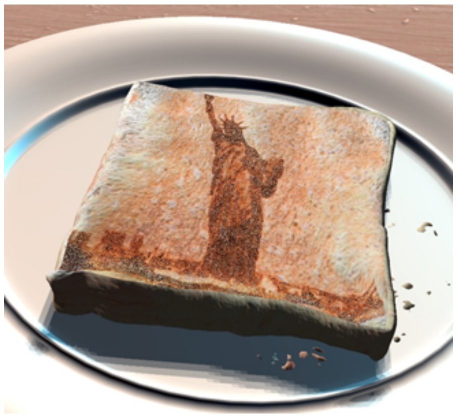
A6 Final Project: Photo Toast



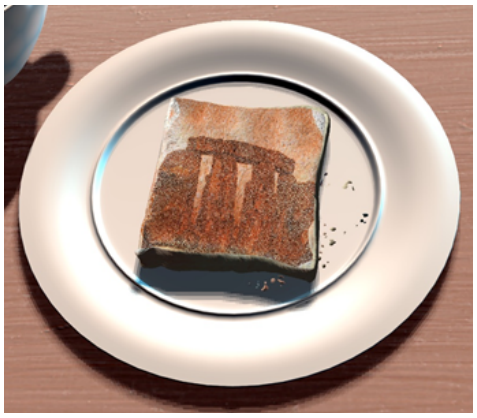
1. Overview

For our project, we were inspired by the “Virgin Mary” toast, a piece of toast that sold for $28,000 because the burnt pattern on the toast resembled a portrait of Virgin Mary. We wondered if we could take any image and burn it onto a piece of toast as if it were naturally occurring.

We present you with Photo Toast: experience a journey from the comfort of your breakfast table!



Statue of Liberty



Stonehenge

1.1 Assets

Here is a list of the assets, i.e., 3D models that appear in our final scene:

1. *background plane*: provided in starter code; applied a texture map
2. *table*: found online; applied a texture map and normal map
3. *cup*: made in Maya; applied a procedurally-generated texture map
4. *plate*: made in Maya; applied a solid color as the texture
5. *toast*: made in Maya; applied a texture map and normal map
6. *crumbs*: made in Maya; applied a texture map

1.2 Lighting

The lighting for our scene is simple but effective. To produce the shadows, we utilized the shadow feature provided in the starter code with the default directional light. For the mug and plate, we used a Phong shader to create a shiny porcelain-like appearance whereas, for everything else, we used the standard Lambertian shader. We tweaked the lighting parameters, e.g., *LightPosition* and *LightIntensity*, for each specific object shader until we were satisfied with the overall aesthetic effect. We focused on making the lighting and scene look as good as possible from a specific range of camera angles.

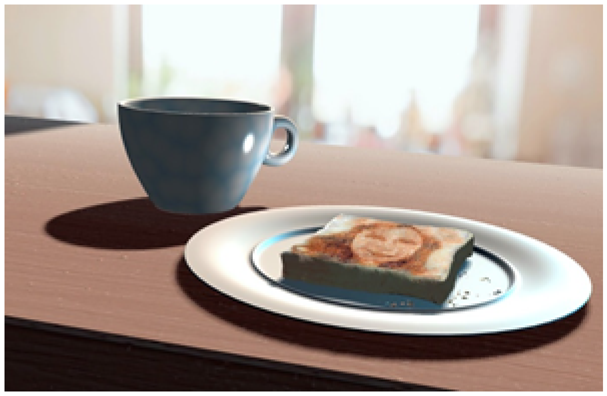
1.3 Image Processing

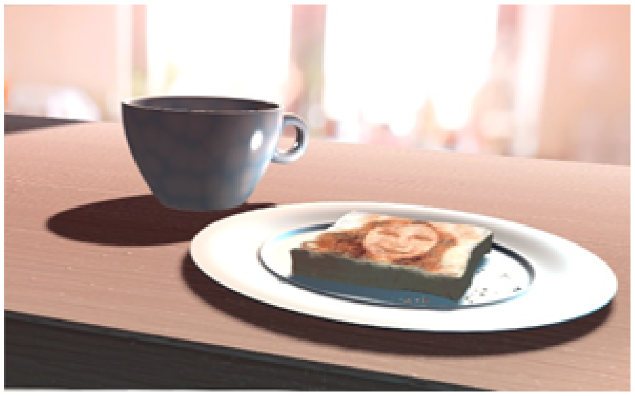
The most significant component of our assignment is image processing, which involves synthesizing a pre-selected image onto the toast in a natural manner and post-processing the final rendered image to add artistic flair.

The image synthesis process is a streamlined procedure that follows a fixed number of steps: blur original image > convert to grayscale > increase contrast > apply Worley and Perlin noise filters > apply heat map.

For post-processing, we loaded the final rendered image onto a plane and applied several aesthetic effects, such as increasing the contrast, adding warmer tones to the lighting, and creating faux depth of field by setting a focal point and blurring texels based on their relative distance from that focal point.

As a disclaimer, for the background image in the original scene, we "cheated" by loading in a texture found online that was already a little blurry. However, we further blurred the background image through code to achieve our intended effect. We also applied a light blur to the coffee cup in the foreground and the far-sided table edge.


Before


After

2. Technical Details

For technical details, we will focus on the image processing techniques and algorithms that were not covered in class or implemented in previous assignments. As mentioned earlier, image processing involves image synthesis and post-processing. For image synthesis, we categorize the techniques as either basic or advanced. All functions related to image synthesis are in "**toast.frag**." For post-processing, we explain the *blurByDistance* function which creates a faux depth of field effect. This function is in "**post.frag**." For the sake of conciseness, we only include short snippets of code. The rest can be found in the appendix.

2.1 Basic Image Synthesis Techniques

Grayscale

We convert the original image to grayscale using the luminosity method, which takes into account the human eye's different levels of sensitivity towards red, green, and blue light. Then, we apply the Hermite cubic function to the resulting gray value to increase the contrast. This can be achieved using the in-built GLSL function *smoothstep*.

```
vec3 grayscale(vec3 color) {
      vec3 coeff = vec3(0.2126, 0.7152, 0.0722);
      float gray = dot(coeff, color);
      return vec3(smoothstep(0, 1, gray));  // increase contrast
}
```

Heat Map

In the final step of image synthesis, we simulate the process of heating the image onto toast using a heat map, which maps the original grayscale colors to a color gradient consisting of colors extracted from real toast.

2.2 Advanced Image Synthesis Techniques

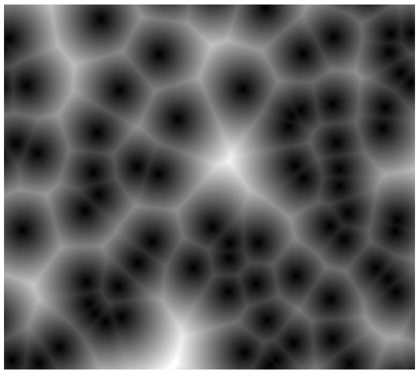
We used Perlin-Worley noise filters to add more realistic texture to the image on toast. To make the image look more natural, we also utilized radial blur.

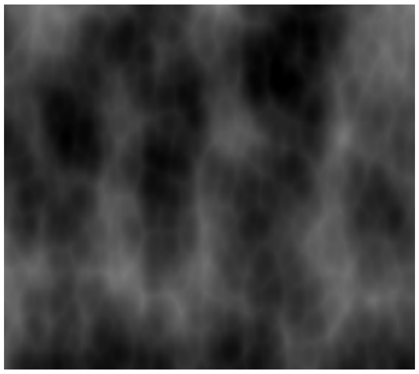
Perlin-Worley Noise

Perlin noise is generated in the same manner as A4 albeit with a different hash function. Worley noise was briefly covered during lecture. It is used to generate noise with a cell-like pattern, based on Voronoi diagrams. To create Worley noise, a random feature point is generated in each tile of the grid. For each point, the noise is the minimum distance to the closest feature point. We only need to look at the feature points in the current tile and the neighboring tiles.

```
float worley_noise(vec2 v)
{
      initialize_var()  // initialize noise, min_dist, idx, fr, etc.

      // find min distance to closest neighbor
      for each neighbor of v:
            // find the feature point in the neighboring tile
            vec2 rand = hash2(idx + offset);
            vec2 nb_pt = offset + rand;

            // compute the distance and update min_dist if necessary
            float dist = length(nb_pt - fr);
            min_dist = min(min_dist, dist);
      }

      // optional: increase contrast, e.g. using Hermite function
      noise = smoothstep(0, 1, min_dist);
      return noise;
}
```

The *noiseOctave* function from A4 is then modified to combine octaves of Perlin noise and Worley noise using a weight parameter.


Worley Noise


Perlin-Worley Noise

Radial Blur

The concept of radial blur is simple: for a given texel, draw a bounding square around the texel, and compute the weighted average of all the texel colors within the bounds.

```
float blur(vec2 uv, int n)
{
        vec3 result = 0.0;

        // n is the radius of blur
        int texels_num = (2*n+1)*(2*n+1);

        // get colors of current texel and neighboring texels
        vec3[texels_num] colors = getColors(uv, n);
        float[texels_num] weights = getWeights(n);

        for(int i = 0; i < texels_num; i++) {
            result += colors[i] * weights[i];
```

```
      }

      return result;
}
```

There are two popular methods for weighting the colors. Box blur applies equal weights whereas Gaussian blur applies weights based on a 2D Gaussian curve. In Gaussian blur, texel colors that are further away from the center contribute much less to the final color. In a manner analogous to bilinear interpolation, the same result can be achieved using 1D horizontal and vertical passes instead of a single 2D pass.

Here is a list of the helper functions that we implemented:

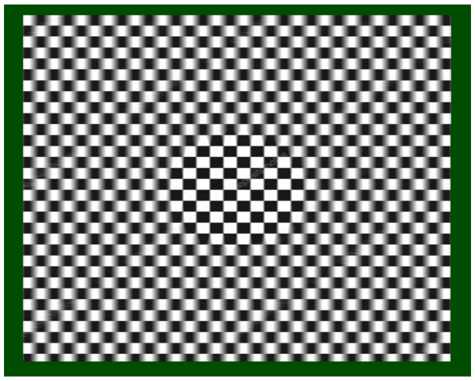
```
// get one row of texel colors
vec3[2*MAX_N+1] getTexelColors(vec2 uv, int n);
// compute binomial coefficient
int binom(int n, int k);
// compute Gaussian weights
float[MAX_N+1] getGaussianWeights(int n);
// compute Box weights
float[MAX_N+1] getBoxWeights(int n) ;
// complete a 1D pass
vec3 blurPass(vec3[2*MAX_N+1] colors, int n, int mode);
```

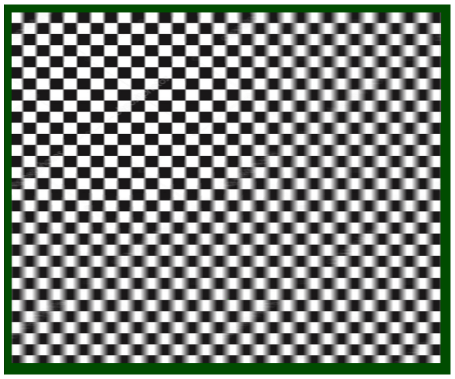
2.3 Post-Processing

To create a faux depth-of-field effect, we wrote a function to blur the final image in the post-processing stage. The function identifies a center of focus and blurs the surrounding texels based on their distance from the focus point. The radius of the blur increases for texels that are further away.

```
vec3 blurByDist(vec2 uv, vec2 uv_center, int n_max, int mode);
```

The input parameters are the uv coordinates of the current texel and center of focus as well as the maximum radius of blur and the weighting mode, i.e., Box or Gaussian. Other parameters within the function determine the size of the area in focus and the sharpness of the boundaries.


Center Focus, Sharp Boundaries


Top-left Focus, Smooth Boundaries

3. Solved Challenges

We encountered many challenges during the course of our project. By solving these challenges, we were able to refine our image processing procedure and significantly improve our final image. Some of the key challenges are solutions are listed below:

Challenge: The image on the toast is difficult to discern.
Solution: Increase the contrast and add a wider range of colors to the heat map.

Challenge: The image on the toast has too many fine details that seem unrealistic.
Solution: Blur the original image before applying the other steps.

Challenge: The texture of the image is too smooth and consistent.
Solution: Apply noise filters to add granular texture and randomness.

Challenge: The blur effect is barely noticeable.
Solution: Increase the blur radius and experiment with different weights.

Challenge: Depth of field is hard to achieve without ray-tracing or extra buffers.
Solution: Create faux depth of field in the post-processing stage using blur.

4. Contributions

Maxine created the objects in Maya, handled the shadows, and implemented Worley-Perlin noise. Winston wrote the grayscale conversion function, the heat map, the radial blur function, and a basic noise filter. Everything else, including the testing and rendering, was done together as a team.

We would like to thank Professor Bo Zhu and all the TA's for providing us with constant support and helpful resources throughout the course of this project.

5. Appendix

Here, we provide our implementations for functions referenced in the "Technical Details" section.

Heat Map (toast.frag)

```
vec3 heat(float val) {
     float w = 0.0;  // color mixing weight
     vec3 c1 = vec3(61,33,27)/255.0;
     vec3 c2 = vec3(91,55,40)/255.0;
     vec3 c3 = vec3(132,76,47)/255.0;
     vec3 c4 = vec3(154,83,56)/255.0;
     vec3 c5 = vec3(189,103,45)/255.0;
     vec3 c6 = vec3(1);

     if (val < 0.2) {
          w = 5 * val;
          return mix(c1, c2, w);
     } else if (val < 0.4) {
          w = 5 * (val - 0.2);
          return mix(c2, c3, w);
     } else if (val < 0.6) {
          w = 5 * (val - 0.4);
          return mix(c3, c4, w);
     } else if (val < 0.8) {
          w = 5 * (val - 0.6);
          return mix(c4, c5, w);
     } else {
            w = 5 * (val - 0.8);
            return mix(c5, c6, w);
        }
}
```

Worley-Perlin Noise (toast.frag)

```
vec2 hash2(vec2 v) {
        vec2 rand = vec2(0,0);

        rand = fract(sin(vec2(dot(v, vec2(127.1, 311.7)),
                    dot(v, vec2(269.5, 183.3)))) * 43758.5453);

        return rand;
}

float perlin_noise(vec2 v) {
        float noise = 0;

        vec2 i = floor(v);  // cell index
        vec2 f = fract(v);  // fraction
        vec2 s = f*f*(3.0-2.0*f);  // weight using cubic Hermite function

        // 2D hash for the four corners of the cell
        vec2 h_a = hash2(i);                    // bottom-left
        vec2 h_b = hash2(i + vec2(1.0, 0.0));  // bottom-right
        vec2 h_c = hash2(i + vec2(0.0, 1.0));  // top-left
        vec2 h_d = hash2(i + vec2(1.0, 1.0));  // top-right

        // 2D position vectors from the four corners of the cell
        vec2 p_a = f;                             // bottom-left
        vec2 p_b = f - vec2(1.0, 0.0);  // bottom-right
        vec2 p_c = f - vec2(0.0, 1.0);  // top-left
        vec2 p_d = f - vec2(1.0, 1.0);  // top-right

        // compute perlin noise at point v using bilinear interpolation
        float noise_bottom = mix(dot(h_a, p_a), dot(h_b, p_b), s.x);
        float noise_top = mix(dot(h_c, p_c), dot(h_d, p_d), s.x);
        noise = mix(noise_bottom, noise_top, s.y);

        return noise;
}

float worley_noise(vec2 v) {
        float noise = 0;
```

```
	vec2 idx = floor(v);  // cell index
	vec2 fr = fract(v);   // fraction

	float min_dist = 10.;  // minimum distance

	float threshold = 0.5;

	for (int j = -1; j <= 1; j++) {
		for (int i = -1; i <= 1; i++) {
			vec2 offset = vec2(float(i), float(j));
			vec2 rand = hash2(idx + offset);
			vec2 nb_pt = offset + rand;
			float dist = length(nb_pt - fr);
			min_dist = min(min_dist, dist);

			// add an additional feature point for some cells
			if (length(rand) > threshold)
				nb_pt = offset + vec2(1) - rand;
				dist = length(nb_pt - fr);
				min_dist = min(min_dist, dist);
		}
	}

	noise = smoothstep(0, 1, min_dist);
	return noise;
}

float noiseOctave(vec2 v, int num) {
	float sum = 0;

	const float scale = 2;
	float a = 1;  // amplitude (f = 1/a is the frequency)
	float w = 0.5;

	for(int i = 0; i < num; i++) {
		sum += w*a*abs(perlin_noise(v/a));
		sum += (1-w)*a*(1-worley_noise(v/a));
		a /= scale;
	}

	return sum;
}
```

Noise Filters (toast.frag)

```
vec3 noiseFilter(vec2 uv) {
	vec2 v = 500 * vec2(uv.x*1.5, uv.y);
	vec3 color = noiseOctave(v, 2) * vec3(1,1,1);
	return color;
}


vec3 getWorley(vec2 uv) {
	vec2 v = 5 * vec2(uv.x*1.5, uv.y);
	vec3 color = noiseOctave(v, 6) * vec3(1,1,1);
	return color;
}

vec3 getPerlin(vec2 uv) {
	vec2 v = 10 * vec2(uv.x*1.5, uv.y);
	vec3 color = 7 * perlin_noise(v) * vec3(1,1,1);
	return color;
}
```

Radial Blur (post.frag)

```
vec3[2*MAX_N+1] getTexelColors(vec2 uv, int n) {
	// get array of texel colors (color of texel at uv, n left neighbors, and n
right neighbors)
	vec3 colors[2*MAX_N+1];
	vec2 offset = vec2(1.0 / textureSize(tex_albedo, 0).x, 0.0);

	colors[0] =  texture(tex_albedo, uv).rgb;  // set color of current texel

	// set color of neighbors
	for(int i = 1; i < n+1; i++) {
		colors[i] = texture(tex_albedo, uv + i * offset).rgb;
		colors[2*n+1-i] = texture(tex_albedo, uv - i * offset).rgb;
		 }

	return colors;
}

int binom(int n, int k) {
    int result = 1;

    if (k > n - k)
        k = n - k;

    for (int i = 0; i < k; i++) {
        result *= (n - i);
        result /= (i + 1);
    }

    return result;
}

float[MAX_N+1] getGaussianWeights(int n) {
      float w[MAX_N+1];
      float total = pow(2.0, 2*n);

      for (int i = 0; i < n+1; i++) {
            w[n-i] = binom(2*n, i)/total;
      }

      return w;
}

float[MAX_N+1] getBoxWeights(int n) {
      float w[MAX_N+1];

      for (int i = 0; i < MAX_N+1; i++) {
            if (i < n+1) {
                  w[i] = 1.0/float(2*n+1);
            }
      }

      return w;
}

vec3 blurPass(vec3[2*MAX_N+1] colors, int n, int mode) {
      float w[MAX_N+1];

      if (mode == BOX) {
            w = getBoxWeights(n);
      } else {
            w = getGaussianWeights(n);
      }

      vec3 color = colors[0] * w[0];  // color contribution of current texel

      for(int i = 1; i < n + 1; i++) {
            color += (colors[i] + colors[2*n+1-i]) * w[i];
    }

      return color;
}
```

```
vec3 blur(vec2 uv, int n, int mode) {
       // based on tutorial: https://learnopengl.com/Advanced-Lighting/Bloom
       // mode == BOX: Box blur
       // mode == GAUSS: Gaussian blur

       vec3 horizontal_colors[2*MAX_N+1];
       vec2 offset = vec2(0.0, 1.0 / textureSize(tex_albedo, 0).y);

       horizontal_colors[0] = blurPass(getTexelColors(uv, n), n, mode);

       for (int i = 1; i < n + 1; i++) {
              horizontal_colors[i] = blurPass(getTexelColors(uv + i * offset, n),
                                             n, mode);
              horizontal_colors[2*n+1-i] = blurPass(getTexelColors(uv - i * offset,
                                                   n), n, mode);
       }

       return blurPass(horizontal_colors, n, mode);
}

vec3 blurByDist(vec2 uv, vec2 uv_center, int n_max, int mode) {
       vec2 size = textureSize(tex_albedo, 0);  // dimensions of image
       vec2 center = uv_center * size;          // coodinates of focus center
       vec2 point = uv * size;                  // coodinates of current point
       float dist = length(point - center);     // distance from focus center
       float max_dist;                          // max distance from focus center
       int n;                                   // chosen blur radius

       if (uv_center.x >= 0.5 && uv_center.y >= 0.5) {
              // center in quadrant I
              max_dist = length(vec2(0.0, 0.0) * size - center);
       } else if (uv_center.x < 0.5 && uv_center.y >= 0.5) {
              // center in quadrant II
              max_dist = length(vec2(1.0, 0.0) * size - center);
       } else if (uv_center.x < 0.5 && uv_center.y < 0.5) {
              // center in quadrant III
              max_dist = length(vec2(1.0, 1.0) * size - center);
       } else {
              // center in quadrant IV
              max_dist = length(vec2(0.0, 1.0) * size - center);
       }

       float f = dist / max_dist;
       float scale = 0.89;  // adjust between 0.87 and 1.00 to change size of focus
       int iter = 10;  // adjust to change sharpness of focus boundary
```

```
        for (int i = 0; i < iter; i++) {
                f = smoothstep(0, 1, f * scale + 1 - scale);
        }

        n = min(n_max, int(n_max * f) + 1);
        return blurPass(getTexelColors(uv, n), n, mode);
}
```