Lab 5: Intro to Textures & Image Processing

Publish | Edit | ⋮

GPR-200: Introduction to Modern Graphics Programming
Instructor: Daniel S. Buckstein
Lab 5: Intro to Textures & Image Processing

Summary:
In lab 4, we explored some ubiquitous simulated lighting models that drastically improve rendering efficiency over raytracing. While this takes us further away from realistic lighting models, we can upgrade our simulated model by introducing textures to modulate surface color. In this assignment we explore different kinds of textures and introductory image processing techniques.

Submission:
Start your work immediately by ensuring your coursework repository is set up, and public. Create a new main branch for this assignment. ***Please work in pairs (see team sign-up) and submit the following once as a team***:

1. Names of contributors
   e.g. **Dan Buckstein**
2. A link to your public repository online
   e.g. **https://github.com/dbucksteinccorg/graphics2-coursework.git**
   (note: this not a real link)
3. The name of the branch that will hold the completed assignment
   e.g. **lab0-main**
4. A link to your video (see below) that can be viewed in a web browser.
   e.g. <insert link to video on YouTube, Google Drive, etc.>

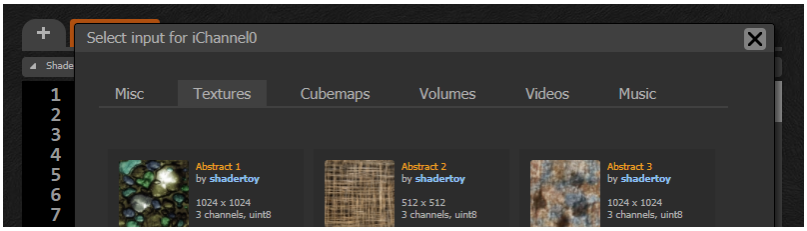
Finally, please submit a ***5-minute max*** demo video of your project. Use the screen and audio capture software of your choice, e.g. Google Meet, to capture a demo of your project as if it were in-class. This should include at least the following, in enough detail to give a thorough idea of what you have created (hint: this is something you could potentially send to an employer so don't minimize it and show off your professionalism):

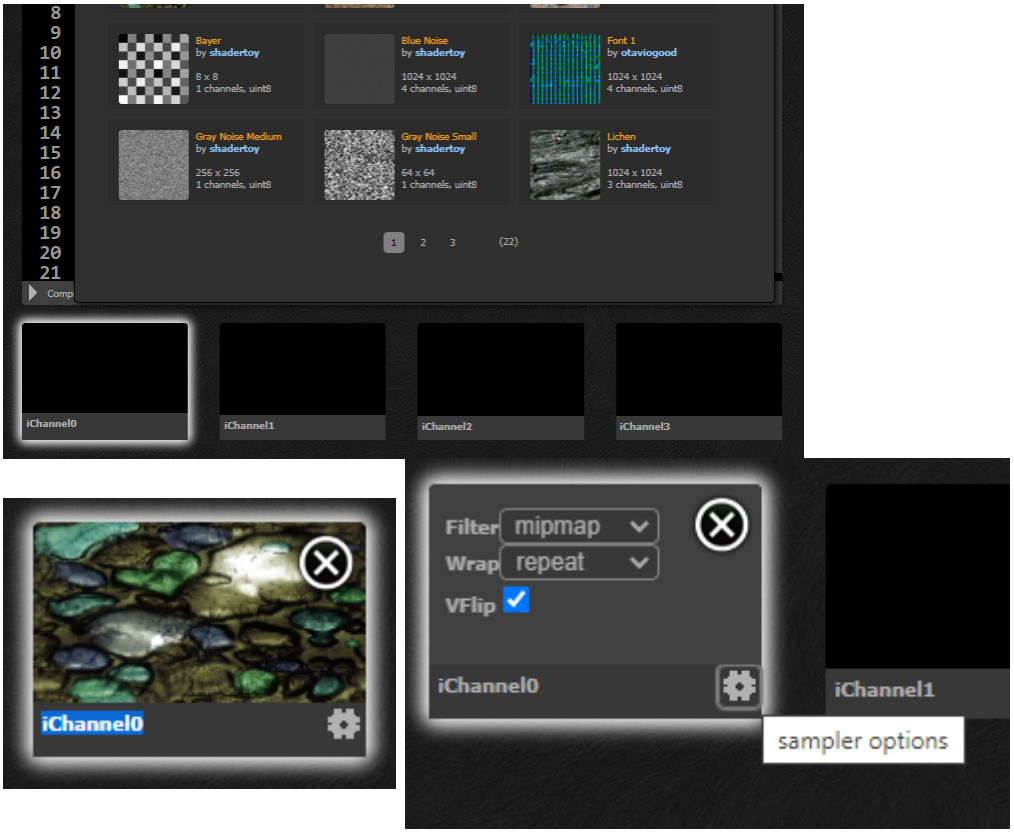
- Show the final result of the project and any features implemented, with a voice over explaining what the user is doing.
- Show and explain any relevant contributions implemented in code and explain their purpose in the context of the assignment and course.
- Show and explain any systems source code implemented, i.e. in framework or application, and explain the purpose of the systems; this includes changes to existing source.
- **DO NOT AIM FOR PERFECTION, JUST GET THE POINT ACROSS**. Please mind the assignment rubric to make sure you have demonstrated enough to cover each category.
- **Please submit a link to a video visible in a web browser, e.g. YouTube or Google Drive.**

Objectives:
The outcome of this assignment is a series of texturing effects, incorporating lighting, image processing and 3D techniques.

Instructions & Requirements:
DO NOT begin programming until you have read through the complete instructions, bonus opportunities and standards, start to finish. Take notes and identify questions during this time. The only exception to this is whatever we do in class.
You may start by copying the contents of **this text file** (https://champlain.instructure.com/courses/1693444/files/190983739/download?download_frd=1) into Shadertoy (download, open in text editor and copy). Using Shadertoy, implement the following texturing effects:

1. ***Cross-fade***: The first required effect explores the fundamentals of textures and getting an interesting cross-fade effect.
   - **Setup:** Shadertoy has a number of built-in textures for your convenience. Here's how to use them.
     - To set up a texture for use in your shader, click on one of the 'channel' boxes located beneath the shader code editor. A window will open with all of the possible inputs. Go to the 'Textures' tab and pick your favorite image. Textures have a number of settings, which we will discuss in class.





- Somewhere in your shader code (e.g. in 'calcColor'), test sampling from the texture by calling the texture sampling function, '*texture*'. The function has two parameters:
  - The first parameter is the ***sampler*****,** which represents the image itself. The value passed comes from the uniform channel uniform that corresponds to the texture channel where you selected your texture. E.g. if you added a texture to the channel farthest to the left, you will see that its uniform name is '*iChannel0*'.
    - For 2D textures, this parameter uses the GLSL sampler type '*sampler2D*'.
    - Pro tip: it is possible to write your own functions that use textures as parameters! Just use the correct GLSL sampler type.
    - ***Note: this is a uniform variable; the texture referenced is the same for all pixels being processed by your shader (i.e. the texture is a consistent input for the entire output image you are displaying).***
  - The second parameter is the ***texture coordinate*****,** the location in the image at which to sample. This is a 2D vector, with each axis containing a value between 0 and 1. This is colloquially referred to as a *UV coordinate*, calculated as some raw pixel coordinate divided by the respective dimensions

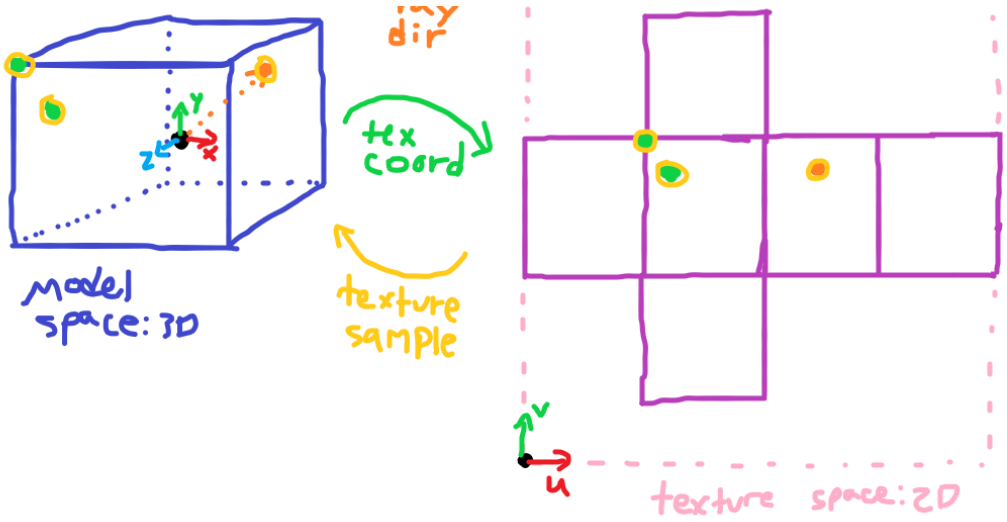
    of the image. To be clear, it is not a unit vector, but the components are within unit range.
    - For 2D textures, this parameter is a 2D vector. Ensure the X and Y components are between 0 and 1 (normalized) to represent the full texture; values beyond this range will produce different results.

        - ***Note: this will be non-uniform as it is dependent on which pixel is currently being processed. Therefore it is derived from the fragment coordinate, e.g. using the UV calculation from earlier labs or something else in the starter code.***
    - Try sampling a variety of textures and coordinate values. You will notice that when using the display's UV coordinate, square textures will appear rectangular. This is because, although the original '*fragCoord*' parameter in '*mainImage*' represents the location of the pixel you are currently processing in the output display, the display pixel coordinate does not necessarily correspond directly to a pixel in the texture. Therefore, to get the correct mapping between the input texture and the display, use one of the '*iChannelResolution*' uniforms (e.g. '*iChannelResolution[0]*' corresponds to '*iChannel0*').
  - **Implementation**: Create a simple cross-fade effect by doing the following:
    - Set up and sample from two or more input channels using a distinct texture for each.
    - Implement a "cross-fade" function that blends the two image samples with respect to time. A blend parameter of 0 results in displaying the first channel image, and 1 results in the second image.
    - Use GLSL's 'mix' function to help (read all about it in the specs).
2. ***Distortion effect***: The next required effect explores modifying the screen-space coordinate, also known as the UV or texture coordinate, to display a distorted image on the scene.
  - **Setup**: This effect follows the same setup as #1.
  - **Implementation**: Once you are comfortable with textures, you will explore basic image processing by creating some image distortion effect. Implement the following:
    - Write a function that takes a 2D pixel coordinate and deforms it using some distortion algorithm (e.g. radial lens distortion: barrel, pincushion, mustache... yes, mustache), or create some interesting pattern with respect to time (waves or ripples). The function should return the distorted coordinate. Regardless of the chosen method, you can implement a trippy distortion effect in only a few lines of code.
    - Convert the distorted coordinate to a UV. Use this to sample the input texture. Because you are using the distorted coordinate instead of the original, this will result in a warping effect on the texture.
    - Demonstrate the difference between the original and distorted image.
3. ***Shading & textures***: The next required effect allows us to implement lighting and shading with textures.

  - **Setup:** This effect follows the same setup as #1.
  - **Implementation**: We will simulate lighting on the image plane instead of on a 3D object (see bonus opportunities if you want to do better with this). Implement the following:
    - *Surface position*: Use either a pixel coordinate (fragCoord) or some other spatial

coordinate (e.g. viewport).
    - *Surface normal*: Use either a constant unit vector pointing along the +Z axis (e.g. normal = vec3(0, 0, 1);), or derived from some unit vector (e.g. normal = normalize(vec3(viewport.xy, 1.0));).
    - *Light position*: Use an arbitrary coordinate with a positive Z value. Make sure it matches the coordinate space of your surface position (e.g. fragCoord is measured in pixels, viewport coordinate is measured in arbitrary scene units).
    - *Lambert shading*: Implement the Lambertian reflectance model using the above information (see lab 4 for a refresher). The only difference is that the surface color comes from a texture.
    - Animate the light's position over time to illuminate different parts of the image.
4. ***Cube map***: The final required effect is to draw a cube map, which can be used as a skybox or backdrop in a 3D scene.
    - **Setup:** The setup for this effect is very similar to #1 with a couple of minor differences.
        - When setting your texture in the channel box, navigate to the 'Cubemaps' tab. A *cube map* texture is a composite of 6 other textures: one for each face in a cube. Imagine the imaginary cube surrounding the viewer (i.e. a skybox). Instead of sampling a 2D image to be applied to a 2D display, a cube map is used to calculate what 3D location on the cube corresponds to the 2D point on the screen.
            - While the '*texture*' function is still used for sampling a cube map, the first parameter's sampler type is now *'samplerCube*'. Pass the channel uniform to the function normally.
            - The second parameter is no longer a 2D vector, but it is now a 3D vector representing the ray direction, much like the one we've been using. If the origin is stationed at the center of the cube, the ray direction is used to determine which face we intersect for the current sample.
    - **Implementation**: Implement a moving cube-mapped image.
        - First, demonstrate sampling from a cube map using the calculated ray direction for the current pixel. You will find yourself looking at the -Z face of the cube map.
        - Animate the ray direction with respect to time (e.g. rotate about the Y axis) to look around the cube map.
        - Apply your distortion effect to the ray direction for a trippy 3D backdrop.
5. ***Optimize***: Per our usual practice, once you make it through the above effects, review and optimize your code.

Figures:
Some more diagrams and notes to help you:



- 3D "model space" is where the surfaces/meshes live. Example of model in Shadertoy: the viewing plane.
- 2D "texture space" is where the images/nets live. Example of an image in Shadertoy: 2D textures, cube maps.
- A color is retrieved for application on a 3D surface by sampling a texture; the 'texture' function returns a color given an image as input and the 2D location at which to sample, known as the UV or texture coordinate.
- Any coordinate in the two spaces can be transformed independently of the other. E.g. the viewing plane never changes, but you can modify your texture coordinate over time to have the texture appear to change over time. You are simply sampling from a different location and applying the result to the same position on the surface.

Bonus:
You are encouraged to complete one or more of the following bonus opportunities (rewards listed):

- ***Wormhole (+1)***: Building on effect #2, implement a black hole or wormhole effect against a scrolling background. The goal is not to make a physically accurate black hole, but one that demonstrates the features in 2D (e.g. event horizon). Do some additional reading as needed.
- ***Procedural shading in 3D (+1)***: Building on effect #3, improve Lambertian/Phong shading on a procedural/ray-traceable 3D sphere, by generating a procedural UV coordinate at each point. This can be achieved by first converting the 3D surface position into a 2D *spherical coordinate*, which represents either longitude-latitude or azimuth-elevation, then convert that to a UV coordinate which can be used to apply a

  texture to the sphere. Stylize a 3D lit and shaded sphere so it appears to have some interesting color and spin like a planet.
- ***Interactive cube map (+1)***: Building on effect #4, add an element of interactivity to your cube map display by controlling the view direction using the mouse. When the mouse button is down, calculate the virtual camera's orientation by converting the click

coordinate to a spherical coordinate (see above), and using that to determine the ray direction for the sampler.
- ***Dynamic lighting (+1)***: Building on effects #3 and 4, implement a spot light that illuminates certain areas of a cube map. Interaction using the mouse should either rotate the cube map or make the light point in a different direction.

Coding Standards:
You are required to mind the following standards (penalties listed):

- ***Reminder: You may be referencing others' ideas and borrowing their code. Credit and provide a link to source materials (books, websites, forums, etc.) in your work wherever code from there is borrowed, and credit your instructor for the starter framework, even if it is adapted, modified or reworked (failure to cite sources and claiming source materials as one's own results in an instant final grade of F in the course)***. Recall that borrowed material, even when cited, is not your own and will not be counted for grades; therefore you must ensure that your assignment includes some of your own contributions and substantial modification from what is provided in the book. ***This principle applies to all evaluations***.
- ***Reminder: You must use version control consistently (zero on organization)***. Even though you are using Shadertoy, a platform which allows you to save your work online, you must frequently commit and back up your work using version control. In your repository, create a new text file for each tab used in Shadertoy (e.g. 'Image') and copy your code there. This will also help you track your progress developing your shaders. Remember to work on a new branch for each assignment.
- ***The assignment must be completed using Shadertoy (https://www.shadertoy.com/) (zero on assignment)***. No other platforms will be permitted for this assignment.
- ***Do not reference uniforms in functions other than 'mainImage' (-1 per instance)***: Use the 'mainImage' function provided to call your effect functions. Any uniform value to be used in a function must be passed as a parameter when the function is called from 'mainImage'.
- ***Use the most efficient methods (-1 per inefficient/unnecessary practices)***: Your goal is to implement optimized and thoughtful shader code instead of just implementing the demo as-is. Use inefficient functions and methods sparingly and out of necessity (including but not limited to conditionals, square roots, etc.). Put some thought into everything you do and figure out if there are more efficient ways to do it.
- ***Every section, block and line of code must be commented (-1 per ambiguous section/block/line)***. Clearly state the intent, the 'why' behind each section, block and

even line (or every few related lines) of code. This is to demonstrate that you can relate what you are doing to the subject matter.
- ***Add author information to the top of each code file (-1 for each omission)***. If you have a license, include the boiler plate template (fill it in with your own info) and add a separate block with: 1) the name and purpose of the file; and 2) a list of contributors and

what they did.

Points 5

Submitting a text entry box or a website url

Due	For	Available from	Until
-	Everyone	-	-

GraphicsAnimation-Master-Range

Criteria	Ratings			Pts
IMPLEMENTATION: Architecture & Design Practical knowledge of C/C++/API/framework programming, engineering and architecture within the provided framework or engine.	**1 to >0.5 pts** **Full points** Strong evidence of efficient and functional C/C++/API/framework code implemented for this assignment; architecture, design and structure are largely both efficient and functional.	**0.5 to >0.0 pts** **Half points** Mild evidence of efficient and functional C/C++/API/framework code implemented for this assignment; architecture, design and structure are largely either efficient or functional.	**0 pts** **Zero points** Weak evidence of efficient and functional C/C++/API/framework code implemented for this assignment; architecture, design and structure are largely neither efficient nor functional.	1 pts
IMPLEMENTATION: Content & Material Practical knowledge of content relevant to the discipline and course (e.g. shaders and effects for graphics, animation algorithms and techniques, etc.).	**1 to >0.5 pts** **Full points** Strong evidence of efficient and functional course- and discipline-specific algorithms and techniques implemented for this assignment; discipline-relevant algorithms and techniques are largely both efficient and functional.	**0.5 to >0.0 pts** **Half points** Mild evidence of efficient and functional course- and discipline-specific algorithms and techniques implemented for this assignment; discipline-relevant algorithms and techniques are largely either efficient or functional.	**0 pts** **Zero points** Weak evidence of efficient and functional course- and discipline-specific algorithms and techniques implemented for this assignment; discipline-relevant algorithms and techniques are largely neither efficient nor functional.	1 pts
DEMONSTRATION: Presentation & Walkthrough Live presentation and walkthrough of code, implementation, contributions, etc.	**1 to >0.5 pts** **Full points** Strong evidence of accuracy and confidence in a live walkthrough of code discussing requirements and high-level contributions; walkthrough is largely both accurate and confident.	**0.5 to >0.0 pts** **Half points** Mild evidence of accuracy and confidence in a live walkthrough of code discussing requirements and high-level contributions; walkthrough is largely either accurate or confident.	**0 pts** **Zero points** Weak evidence of accuracy and confidence in a live walkthrough of code discussing requirements and high-level contributions; walkthrough is largely neither accurate nor confident.	1 pts
DEMONSTRATION: Product & Output Live showing and explanation of final working implementation, product and/or outputs.	**1 to >0.5 pts** **Full points** Strong evidence of correct and stable final product that runs as expected; end result is largely both correct and stable.	**0.5 to >0.0 pts** **Half points** Mild evidence of correct and stable final product that runs as expected; end result is largely either correct or stable.	**0 pts** **Zero points** Weak evidence of correct and stable final product that runs as expected; end result is largely neither correct nor stable.	1 pts

Criteria	Ratings			Pts
ORGANIZATION: Documentation & Management Overall developer communication practices, such as thorough documentation and use of version control.	**1 to >0.5 pts** **Full points** Strong evidence of thorough code documentation and commenting, and consistent organization and management with version control; project is largely both documented and organized.	**0.5 to >0.0 pts** **Half points** Mild evidence of thorough code documentation and commenting, and consistent organization and management with version control; project is largely either documented or organized.	**0 pts** **Zero points** Weak evidence of thorough code documentation and commenting, and consistent organization and management with version control; project is largely neither documented nor organized.	1 pts
BONUSES Bonus points may be awarded for extra credit contributions.	**0 pts** **Points awarded** If score is positive, points were awarded for extra credit contributions (see comments).		**0 pts** **Zero points**	0 pts
PENALTIES Penalty points may be deducted for coding standard violations.	**0 pts** **Points deducted** If score is negative, points were deducted for coding standard violations (see comments).		**0 pts** **Zero points**	0 pts
				Total Points: 5