1. Neon Pool

Part A:

1. Design a 3D scene with objects of your choice. The scene should contain spheres, which will resemble billiard balls. Keep in mind that later in the project these spheres will act as light sources.
2. Place a central light source on the scene and implement lighting and shading algorithms.
3. Place a second light source of the spotlight type. This light source should be placed at an angle to create interesting shadows. Also, this light source should stand out as a spotlight, as it will only illuminate a part of the scene.

Part B:

4. Implement the deferred rendering method. Using this method you can insert multiple light sources.
5. Convert the spheres, which you inserted in the first task, into independent light sources. Each sphere should have different lightning lightning properties, such as color. Find a way to be able to pass the properties of each light source to the shaders, which way can be configured from code or during runtime. (This should not be hardcoded).
   Note that for these lighting sources no shadow computations will be done. These spheres, however, block light caused by other sources. That is, if a sphere is placed very close to the spotlight, which has a higher intensity, the shadow of the ball will be visible.
6. By pressing a button we should see the top view of the scene. Here, create an interface where the user can define the direction and the initial velocity of the ball.
7. Implement the motion of the sphere. The sphere will move in space and collide with the rest of the spheres that will be in the scene.
8. Implement a *smart* algorithm to handle collisions. For instance, there is no need to check collisions between balls that are far apart.

Bonus:

- Implement the Screen Space Ambient Occlusion technique for more realistic scene shading.
- Convert the first light source to a Point light, implementing the appropriate shading method as well.
- Add a menu using the IMGUI library.
- Enable the user to dynamically change the number of balls from the interface.
- Use the instancing method to render spheres more efficiently.

2. Portal Maze

Part A:

1. Create cube models with various textures, which could be used to create the walls of a maze. Create an algorithm that will use these walls to construct a random maze.
2. Apply lighting and shading techniques to illuminate the scene.
3. Create an avatar to represent the character. When the user enter the game they will be in first person. That is, the camera will be placed at the character's head and will move based on the limitations of that character. By pressing a button, the camera will be able to detach from the character and move around in space as well as view the scene from above. By pressing a button again, the camera will return to the character.

Part B:

4. Make use of the instancing technique to efficiently place the cubes.
5. Create a portal gun. Using a button you will use the portal gun to open a portal on a target. Once a pair of portals is created, the user will be able to see through one of the portals to the scene on the other side. To create the portals you can use the stencil buffer method.
6. Allow the user to pass through portals and be on the other side. Modify the generation algorithm of the maze to contain small holes, in such a scale in order for the user to look through them but not be able to pass through, so that they can open a portal on the other side.

Bonus:

- Implement physics simulations for objects passing through portals.
- View from above the moment an object is halfway through the portal.

3. Retro Search

Part A:

1. Create a small city. Generate the ground and load models of different buildings. Color them all different (solid) colors or load textures for them. Make sure that one of the buildings stands out (make it very big, interesting-looking, or color it in an eye-catching way).
2. Implement collision. The player should not be able to walk through the building walls. If the buildings have doors, you can make it so the player can walk inside the buildings using them. But leave the door of the eye-catching building closed.
3. Load a model of a key and a lock. Place a few locks on the door of the locked building and hide the keys around the map. Have it so each time you run the game, the keys are hidden in different spots. Have the player "pick up" a key every time they touch one. If they have collected a key and they touch one of the locks, that locks should disappear. When all the locks are removed the player gets access to the building and the game ends.

Part B:

4. Implement any form of shading and shadows you want.
5. Create a "scanlines" effect and make the game look like it's being viewed through a CRT monitor.
6. Limit your color palette to only a few colors (2,4, or 8, no more). Only these colors should be rendered, any other colors should be converted to the closest of these.
7. Implement dithering to improve the end result.

Bonus:

This game is now your own. Make any improvements you want. Include a HUD that shows how many keys you have collected or add realistic walking. There are no bad ideas!

4. Monument Valley

Part A:

1. Develop a 3D stage. Create paths in different height levels using voxels that would guide to a specific goal (e.g. a monument, a statue etc)[a]. Try to create inclined paths and obstacles. Use various textures to decorate your stage.
2. Create a sphere that will pose as the player's avatar and add physical properties to it (e.g. gravity so that it would "slide" down the path).
3. Specify the movement of the sphere so that it would be able to reach the goal. Add a particle effect that it would be activated upon reaching the goal.
4. The player should be able to rotate the camera around the stage.

Part B:

5. Add lightning and shadows. Use any of the common techniques.
6. Make parts of your path interactable. The player should interact of the path to find hidden passages in order for the sphere to reach the target through these new passages. It would be desirable so that all paths are hidden and no direct solution is visible, so that the user has to manipulate the stage. The movement of the sphere should be enabled when the puzzle is solved.
7. Add a highlight effect when a part of the path is selected by the user.

Bonus:

Replace the sphere with a humanoid avatar and animate its movement. The player should be able to move the avatar as they like.

[a] Examples of the desirable design of the puzzle stages - https://www.monumentvalleygame.com/mv1

5. Waves

Part A:

1. Create a grid with a texture that would represent our sea.
2. Implement multiple texture wave illusion.
3. Add a sinus function to animate a wave.
4. Add a second sinus function to animate two waves. You can try to add this wave in the opposite direction than the first one.

Part B:

5. Add more waves to the sea. You can use the Gerstner [b] method to create the effect of waves intersecting with each other.
6. Add lightning and shadows. Use any of the common techniques.
7. Add particles to simulate the sea foam and bubbles when waves are intersecting with each other.

Bonus:

Create a reflections using the Fresnel effect [c]

[b] Gestner waves - https://en.wikipedia.org/wiki/Trochoidal_wave
[c] Fresnel effect - https://en.wikipedia.org/wiki/Fresnel_equations

6. Art gallery

Part A:

1. Create a cylindrical room with 5 paintings equally spaced around the wall. Each painting will be associated with a different room/scene. The user can enter the painting and get into the respective room.
2. Create the rooms, add objects, lighting and shadows.
3. Each room will have a unique art style. The art style will be applied on the resulting frame, before showing it to the user. Create 5 separate functions/shaders that will handle the frame buffers of each of the different rooms.

Part B:

4. a. Room 1: Implement Floyd-Steinberg dithering, to quantize the colors in the frame.
   b. Room 2: Create a brush stroke effect.
   c. Room 3: Instead of solid color, use small circles of various sizes (Pointilism)
   d. Room 4: Implement a grayscale, comic book effect.
   e. Room 5: Choose a different style according to your preferences.
5. Generate a bumb map based on the normals of the scenes and use them for bump renderings of the paintings
6. Use a depth map to raise the mesh of the paintings and generate an anaglyph representation

Resources:

- https://en.wikipedia.org/wiki/Floyd%E2%80%93Steinberg_dithering