1. Winter

Part A:

1. Load a mountainous terrain (or generate one, if you prefer). Make sure there is a valley amidst the mountains. Add clouds in the sky and make them look as realistic as you can.
2. Inside the valley, create a small lake. Make the water look realistic by using multiple water textures with displacement (like we did in lab 3).
3. Add a directional light source to act as the sun. Make sure to adjust the appropriate light properties to make the light fit in a winter setting (e.g., blueish color tones, low intensity). Also add shadows using any technique you want.
4. Add sparse vegetation (patches of grass, short bushes, etc.).

Part B:

5. Create the effect that it’s snowing. The snowfall is initially disabled and starts with the press of a key on the keyboard. While the snowfall continues, snow builds up on top of the terrain and the landscape becomes “snowy”. The lake freezes and the water turns into ice. The environment is reflected atop the ice lake’s surface.
6. By pressing a second key, strong wind starts blowing, affecting the snowfall and vegetation movement.
7. Create the following effects:
   a. With the press of a third key, the clouds start to clear away and the sun comes out. Adjust the light properties accordingly.
   b. The ice in the lake appears to crack and melts back into water.
   c. The snow that was collected in the environment begins melting and starts to roll downhill, filling the lake further.
   d. As the melting of the snow accelerates, an avalanche is caused.

Bonus:

1. Add a screen-space trembling/shaking effect that happens for a few seconds each time after the player camera touches snow, ice, or otherwise cold surfaces.
2. Make the water look even more realistic by modeling more advanced effects. You can model reflection/refraction, the Fresnel effect, realistic waves with DuDv maps or Gerstner waves, or any other effect you see fit.

2. Roaming Snail

Part A:

1. Use a heightmap to create a randomly generated terrain that includes hills and valleys.
2. Create or find a 3D model of a snail as the player character. The snail should move along the terrain by "sticking" to the environment. Use mesh manipulation to animate the snail. The motion of the snail should be synchronized with creeping-like animation.
3. In its current state, the snail can climb steep surfaces by sticking to them but moves slowly. Make it so, that when the user presses a button, the snail can retract into its shell (create an animation for this)
4. When the snail is retracted, make it so the shell can roll along the environment much faster, but can't climb steep surfaces. Make it so the shell obeys the laws of Newtonian mechanics when the user is not applying control (conservation of momentum, etc.). When the user moves the shell, it should provide some power, but not enough to climb steep surfaces, for example.

Part B:

5. Implement friction between the shell and environment and implement spin.
6. Texture the terrain. Apply random rotation/flipping to the textures to eliminate repetitiveness.
7. Implement any lighting techniques you want and create a skybox
8. Populate the environment with vegetation. When the snail in its shell form collides with vegetation, It slows down and the motion of the vegetaion is affected. When in its regular form it can eat it and can move faster for a time.
9. Make the interaction between the terrain and the retracted snail be as fun and intuitive as you can. The snail should be able to slingshot over hills when in high speeds, and bounce when colliding with the ground, without the system reaching instability.

3. UP – The Remake

Part A:

1. Create a 3D house and a good amount of balloons (at least 8) that will be attached (e.g. on its roof) using strings. Moreover, develop a rocky/desert-like environment using any method you prefer (e.g. heightmap) and place the house on the top of the highest hill.
2. Implement basic physics for the 3D objects in the scene (e.g., gravity, collision detection). For example, the balloons should not be penetrating one another.
3. Create a different characteristic for each balloon:
    a. A balloon with glitter (preferable on its surface),
    b. A balloon with a metallic finish,
    c. A neon balloon,
    d. A transparent balloon that will contain a small scene or a 3D object inside it,
    e. A balloon with 3D texture, etc.
4. Implement basic lighting and shadows based on any of the common techniques learned in the lab.

Part B:

5. Implement a flight simulation algorithm for the house. Specify a target space in the environment and make the house fly until the target. Make the house drift in the air while it flies, and also simulate the resistance due to air.
6. Create "enemies" (e.g. birds), which will make the balloons pop upon collision. Create a popping effect using particles or any other method. The house should lose height proportionally to the balloons that have been destroyed.
7. Develop a user control mechanism to control the trajectory of the house while flying.
8. Make all balloons pop simultaneously. The house should fall with an increasing speed towards the ground. Upon contact with the ground, make the house fall apart into wooden pieces.

Bonus:

1. Make the string of a popped balloon fall realistically, while being attached to the house.
2. Add 3D characters in the scene (e.g. people, animals), and enable the user to attach balloons in their back. Make the characters fly in a height that is proportional to the balloons that they have (e.g. 2 balloons make the character fly higher, than 1, etc.).

4. Elemental

Part A:

1. Create a 2D plane and use it to design a volcano. Add a night sky, stars and clouds to the scene. Add an empty river at the base of the volcano.
2. Add lava inside the volcano and make it run realistically down the slope. When the lava will reach the base of the volcano it will fill the empty river. Make the lava glow.
3. Develop lighting and shadow algorithms based on the techniques learned in the lab.

Part B:

4. Create a domino effect of elemental transformations:
    a. When the lava reaches the bottom of the volcano, it will transform into smoke and ashes.
    b. The smoke will form into clouds that will produce rain.
    c. The rain will fill the river with water.
    d. When the water reaches the end of the river, it will transform into flowers and trees.
5. Create a time controller interface that will speed up or slow down simulation-time speed
6. Add some lightning bolts that will stem from the clouds and fall on the ground. Upon touching the ground, they will leave burned marks on the terrain.
7. Mix two elements together to create a physical or a fictitious effect. For example, you may blend air with earth and create a tornado that will contain ground particles. Try to make the transitions (from element blending to effect) as smooth as possible.

5. Real Time Strategy

Part A:

1. Create a large terrain. Add objects (trees, rocks etc) such that the terrain is symmetric. On two opposite sites of the map there will be two player bases.
2. Implement lighting and shadow algorithms. Modify the camera object so that the player can view the terrain from above (as if from the sky).
3. Create the game content:
    a. At least 2 resources (food and wood)
    b. At least two types of buildings (town center and barracks)
    c. At least 3 types of units (2 military; one ranged and one melee and 1 economic unit). The former spawn from the barracks, whereas the latter spawns from the town center.
4. Implement unit behavior. The military units fight enemy units, the economic unit gathers from natural resources. Create an indication for when they do their respective tasks, e.g. their colors change.
5. Make the building construction gradual. Start from a mesh of the building foundation and fade in/out towards the final building mesh, or cut the final mesh into arbitrary parts and gradually show each one until the building is complete.

Part B:

5. Implement ray casting and drag-box to select your units on the map. Implement a camera change feature, with which you can view the world from the point of view of a selected unit.
6. Implement a path finding algorithm to guide units towards their target while avoiding object
7. Handle the rendering/physics effectively, as it will be possible to have a lot of units on the map. Use frustum culling to render only the part of the map that the camera can see. Use instancing to handle the plethora of units.
8. Make melee units explode on command, deforming the terrain permanently. These deformations should affect the way the path finding algorithm functions, adding the area to the list of obstacles.

Bonus: You may implement other features typically found in RTS games, such as a GUI to more efficiently manage your units, fog of war, projectiles, basic unit AI, progress bars above each unit to indicate its health (programmed as 2D elements, or 3D elements that always face the camera), etc.

Additional details: RTS games expect the player to create and micromanage multiple units simultaneously using strategy and tactics. Some actions, such as the creation of units, cost resources. The town center spawns economic units which gather from natural resources on the map and military units fight. The player who destroys the enemy's town center wins. All units and buildings have stats, such as HP, damage etc. Manage these effectively. You can add animations if you like (for fighting, gathering etc), but they are not necessary. However, make it easy for the user to understand what each unit is doing. TIP: start small and build up as you go.