CG1 WS 20/21 - Exercise 2: Canonical View

Technische Universität Berlin - Computer Graphics
Date 26. November 2020 **Deadline** 09. December 2020
Prof. Dr. Marc Alexa, Ugo Finnendahl, Max Kohlbrenner

Projection / Canonical viewing volume (8 Points)

This exercise is built on the Nate Robins OpenGL projection tutorial and it extends the visualization by the canonical viewing volume (normalized device coordinates).

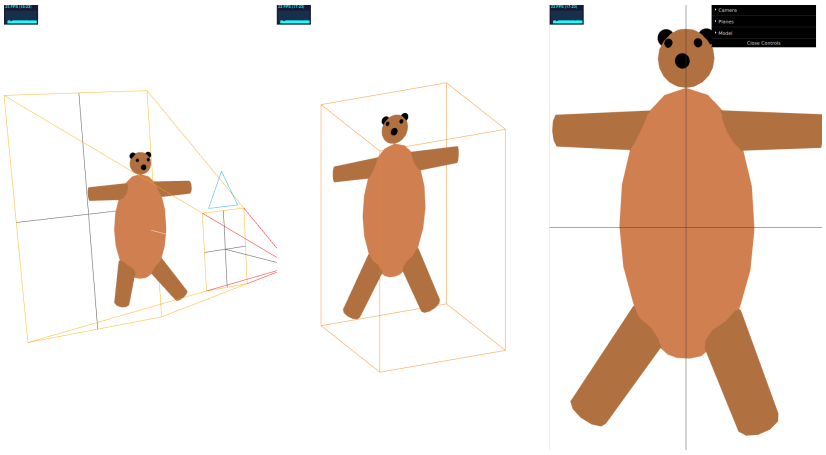

Figure 1.1: The scene in world space (left), canonical viewing space (middle) and screen space (right).

The usual way to visualize a 3 dimensional scene on a two dimensional output device such as a computer screen avoids simulating the physics of light. Instead it uses a projective transformation of the scene to simulate a camera. In this exercise, you explore the projection step that is needed to transform a scene onto the screen and visualize the effect of different camera parameters. Figure 1.1 shows an example view of the three different spaces you are working in. The screen space (right) is the projected space as viewed by the camera, the world space (left) shows the scene including the camera that is used for the rendering of the screen space and the region effectively viewed by the camera called the frustum. In the process of transforming the world space to screen space, an intermediate representation in normalized device coordinates (NDC, middle) is used. By applying the cameras perspective projection, it effectively transforms the cameras frustum to a cube called the canonical viewing volume. The perspective transformation depends on the camera prameters and enlarges the parts of the scene that are closer to the camera . The canonical viewing volume facilitates clipping the visible part of the image and can be easy transformed to screen space by a second, parallel, projection. Note that the rendering of the cube does not have the same height and width, this is due to the convention of the window coordinate system, nevertheless all edges have the same length in the NDC coordinate system.

In this exercise sheet, you render the three different views and update them accordingly in case internal camera parameters, viewing angle or object position has been altered using the GUI or orbit controls.

The tasks in detail are:

1. **Basic Setup:** Divide the application in three windows with equal size, that will be filled later and create the gui (`Settings` and `createGUI` are provided in `helper.ts`). (*0.5 point*)

2. **Screen space (right):** Create a scene with a white background color containing a teddy bear (the object creation is provided by `createTeddyBear` in `helper.ts`) and render it in the right window using a perspective camera (from now on called screen camera). That camera must be controllable by `OrbitConrols`. Implement the gui functionalities in the model folder, so that you are able to translate and rotate the teddy bear in all three dimensions. (*1 point*)

3. **World space (left):** Render the same scene using a different perspective camera (now called world camera) that is also controllable using `OrbitConrols`. Implement the gui functionalities in the camera folder, so that the screen camera parameters are updated accordingly. Add a visualization of the screen camera (e.g. using `CameraHelper` in `Three.js`) to the scene. The visualization is also updated whenever the screen camera is changed (either by parameter changes or by transformation changes introduced by the `OrbitControls`). Consequently we can observe the corresponding updates in both world space (by the camera visualization) and screen space (different output). *Hint:* First: Whenever you change the camera properties you have to call `updateProjectionMatrix` to that the projection matrix is updated according to the parameters. Second: You can listen on `OrbitControl` changes using `addEventListener` with the `'change'` event. That event is triggered whenever the controls change the transformation of the camera. (*1.5 point*)

4. **Normalized device coordinates (NDC, mid):** Create a new scene (again with a white background) that is render in the mid window using an orthographic[1] camera (provided by `Three.js`) that is again controllable using `OrbitControls`. Add a cube with length 2 to the new scene that is centered at the origin (provided by `setupCube` in `helper.ts`). Add a copy of the original teddy bear to the scene and implement a function that transforms the teddy bear into the canonical viewing space (normalized device coordinates, NDC). That function needs to be called whenever the camera or the teddy bear changes. (*4 points*)

   Restriction: For this task, you are not allowed to use the `applyMatrix4` neither of the `Geometry` nor the `VectorX` for `X` $\in \{3, 4\}$ class. Iterate over the vertices of the geometry and transform each vertex individually and apply the perspective transformation by hand. Vertex positions can be accessed using the `vertices` property of the `Geometry` class. Make sure that `verticesNeedUpdate` is set to true after the change.

   Hints:

   - The camera coordinate system is left-handed. Since the rendering of the NDC is itself done by a left-handed camera, you need to invert one of the axes of the NDC to show it in the correct orientation.
   - The transformation from local to the global coordinate system you need to calculate last exercise is provided by the `Object3D` class in the `matrixWorld` property. This time you are allowed to use it, but you can use your previous implementation.
   - In `Three.js`, the camera transformation from local to global coordinates is stored similar to an `Object3d`. Beside `matrixWorld` it also provides its inverse in `matrixWorldInverse` and the projection matrix in `projectionMatrix`. Use the `updateMatrix`, `updateMatrixWorld` and `updateProjectionMatrix` methods to make sure that they are up to date whenever the camera parameters changed.
   - `Object3D.traverse` can largely simplify the computation.

5. Add six clipping planes to simulate the canonical viewing volume (i.e. anything outside the viewing volume is removed). Extend the GUI so that the clipping planes can be toggled on and off. *Hint:* You can add clipping planes to a renderer using the `clippingPlanes` property of the `renderer` object. (*1 point*)

Requirements

- Exercises must be completed individually. Plagiarism will lead to exclusion from the course.
- Submit a `.zip` file of the `src` folder of your solution through ISIS by **09. December 2020, 23:59**.
- *Naming convention*: {firstname}_{lastname}_cg1_ex{#}.zip (for example: jane_doe_cg1_ex2.zip).
- You only hand in your `src` folder, make sure your code works with the rest of the provided skeleton.

[1] Trivia: An orthographic camera is not necessary but enables us to produce the same output in the right and mid rendering.