Towards a Fully Customized Renderer for MecSimCalc

Ziyu Sun[1], Samer Adeeb[2]

[1]Jilin University, Changchun, China, [2]University of of Alberta, Edmonton, Canada

Research Background

Creating a renderer of full control, flexibility and scalability for MecSimCalc.

MecSimCalc, developed by the University of Alberta, allows global users around the world to easily share and contribute Python codes and applications. It lets developers create customizable tools for specific needs that others, including students and researchers, can use without any code knowledge. This platform improves the sharing of ideas and codes in education and research.

Graphic renderer translates virtual content onto the screen, forming the foundation of the digital world. By developing a customized, controllable, and easily expandable renderer, we can unlock advanced visualization features such as 3D rendering and interactive gaming experiences. For research and educational purposes, **we are creating this renderer from scratch, without relying on third-party python packages**. This approach ensures full control, flexibility, and scalability, as we handle every step of the rendering process—from constructing triangles to fragment.

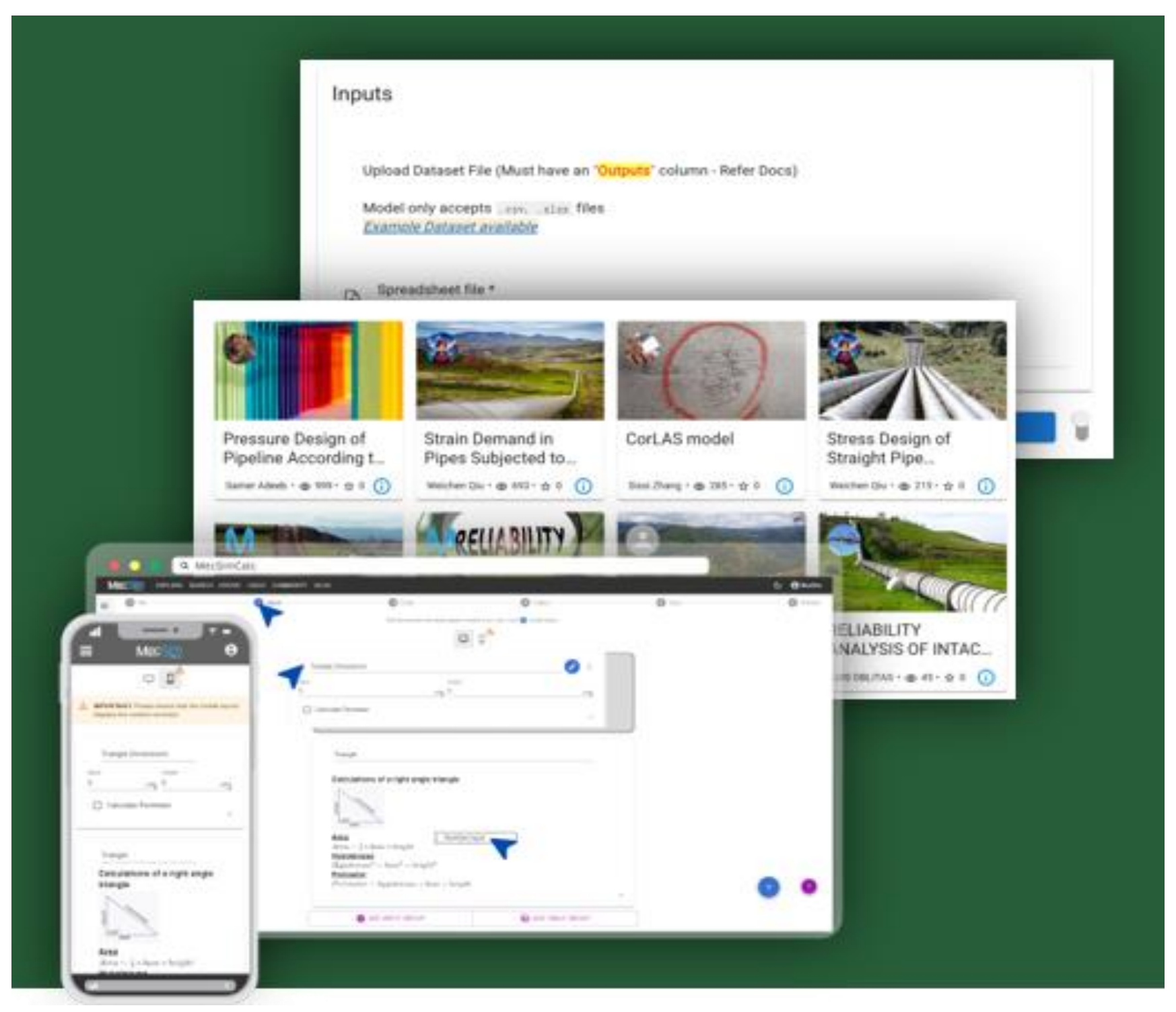


Fig. 1: MecSimCalc: A platform to create and share codes.

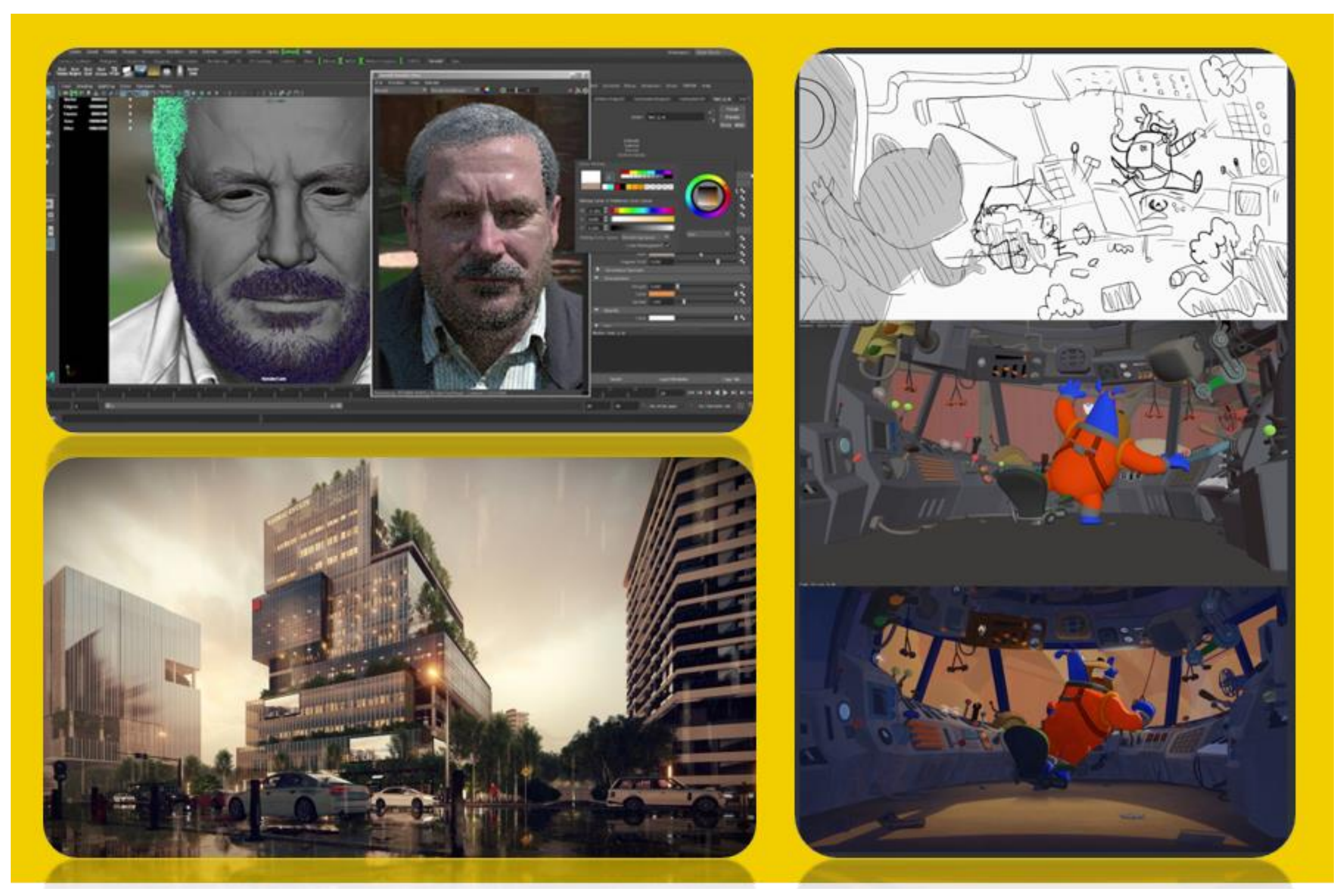
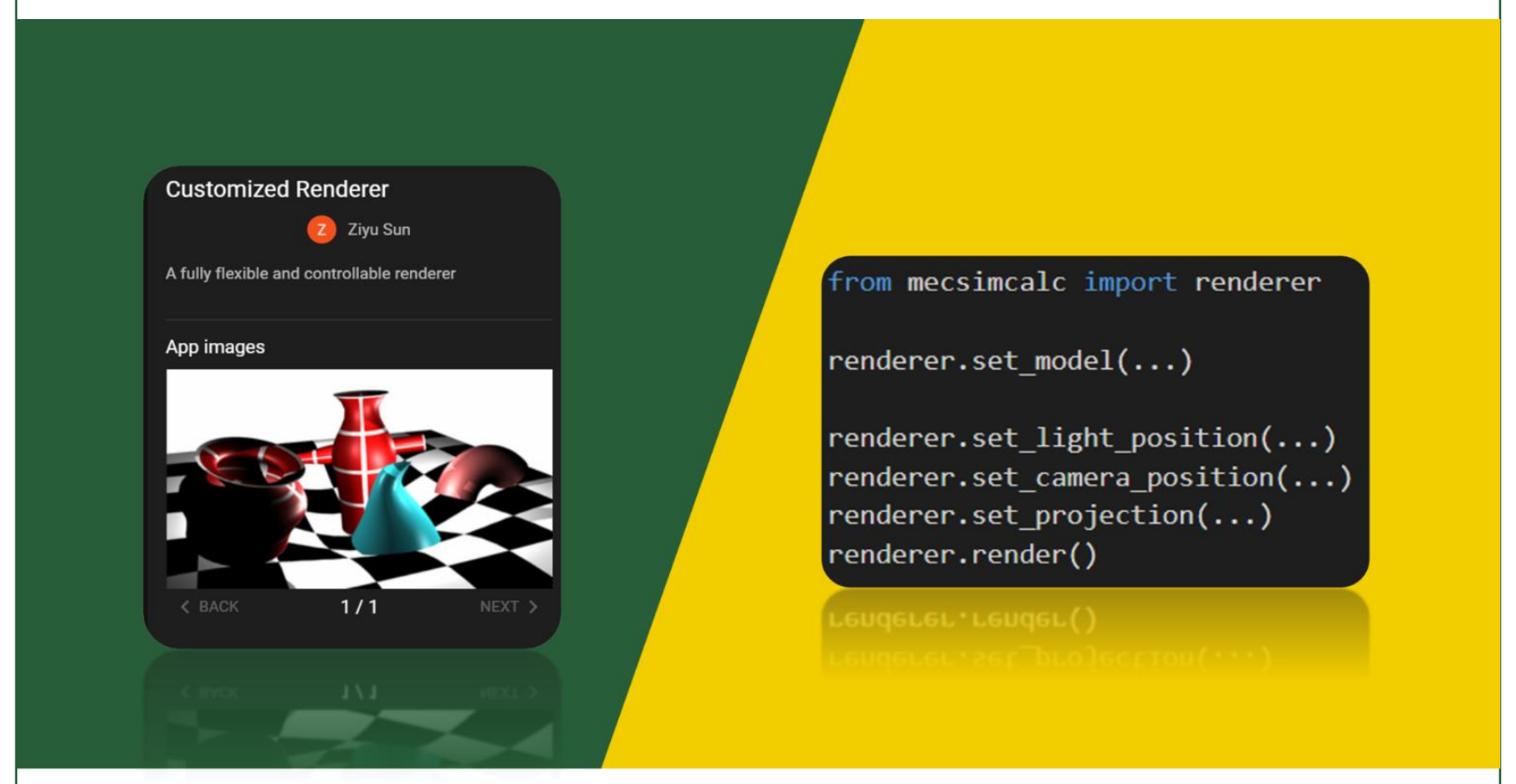


Fig. 2: Renderer in Game Engines.

Application

For users, for developers.

The customized renderer serves two main purposes:



```
from mecsimcalc import renderer

renderer.set_model(...)

renderer.set_light_position(...)
renderer.set_camera_position(...)
renderer.set_projection(...)
renderer.render()
```

Fig. 4: Application / package.

First, it allows the creation of a graphic pipeline to build user-friendly Python applications for rendering 3D assets in a flexible and controllable manner.

Second, it will be integrated into the MecSimCalc package to handle downstream rendering tasks that may not be easily managed by black-box renderers.

Rendering Results

Achieving outstanding results step by step.

Below are the partial rendering results.

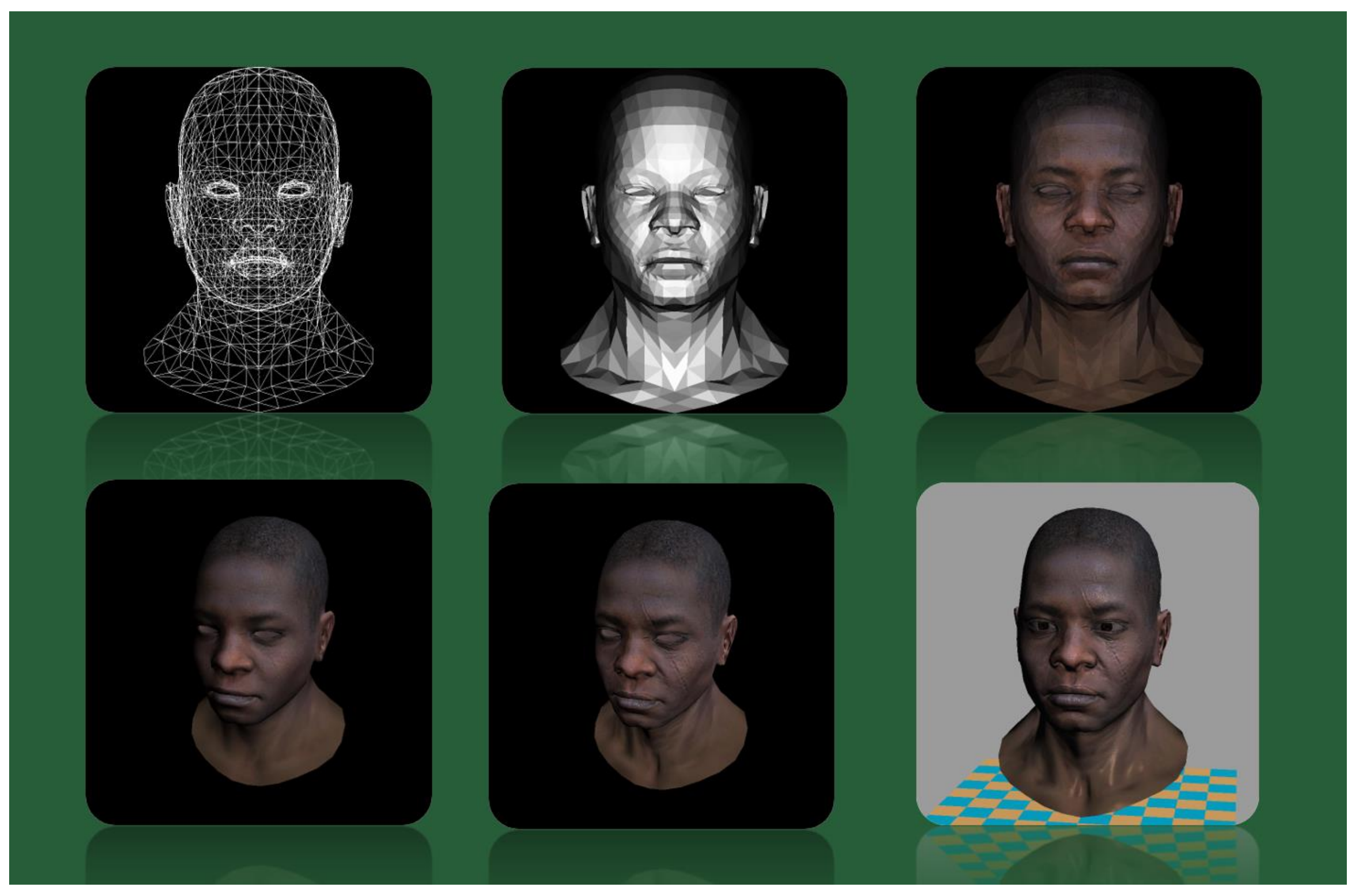


Fig. 5: Rendering results step by step.

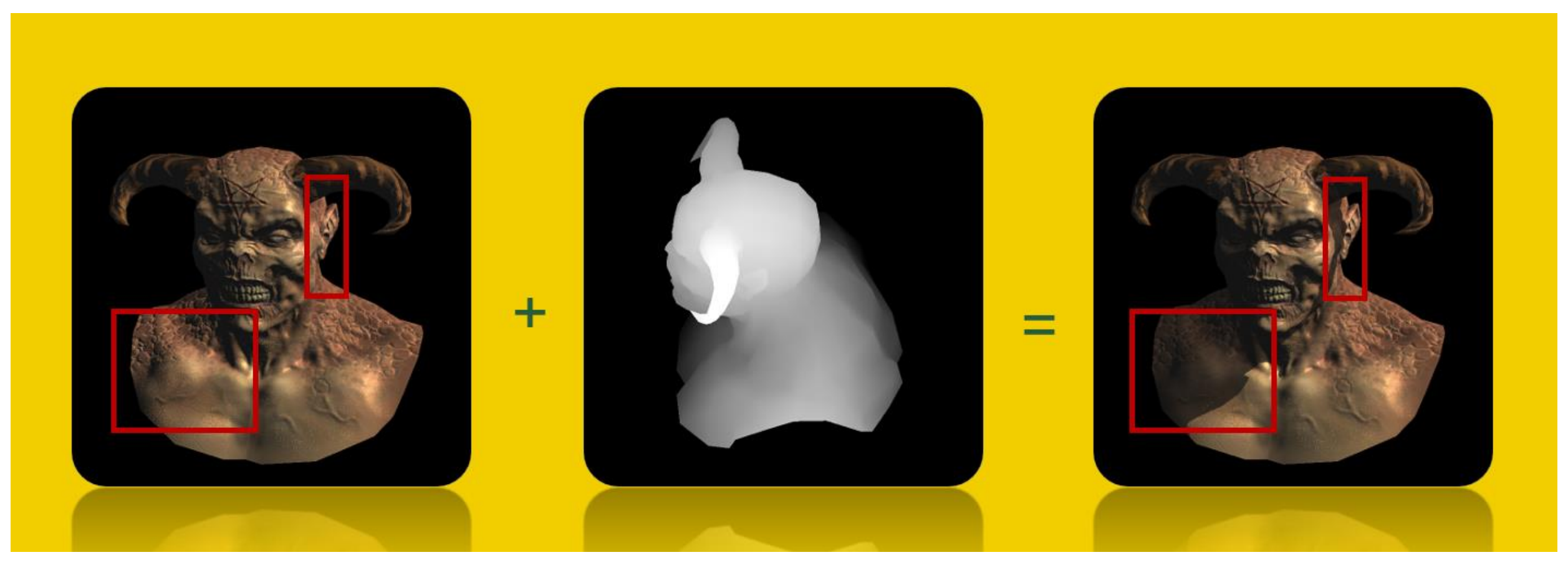


Fig. 6: A visualization of shadow mapping.

Technical Solution

Implementing the complete pipeline of a classic renderer.

We follow the classic graphics pipeline, typically consisting of 7 stages.

Application: We set up the scene in this stage, including the creation of models, setting camera views, and specifying lighting and materials.

Command: We buffer and interpret commands in this stage.

Geometry: In the Geometry Stage, we perform transformations and projections from object to image space, handle clipping, culling, and primitive assembly, calculate lighting effects and generate texture coordinates.

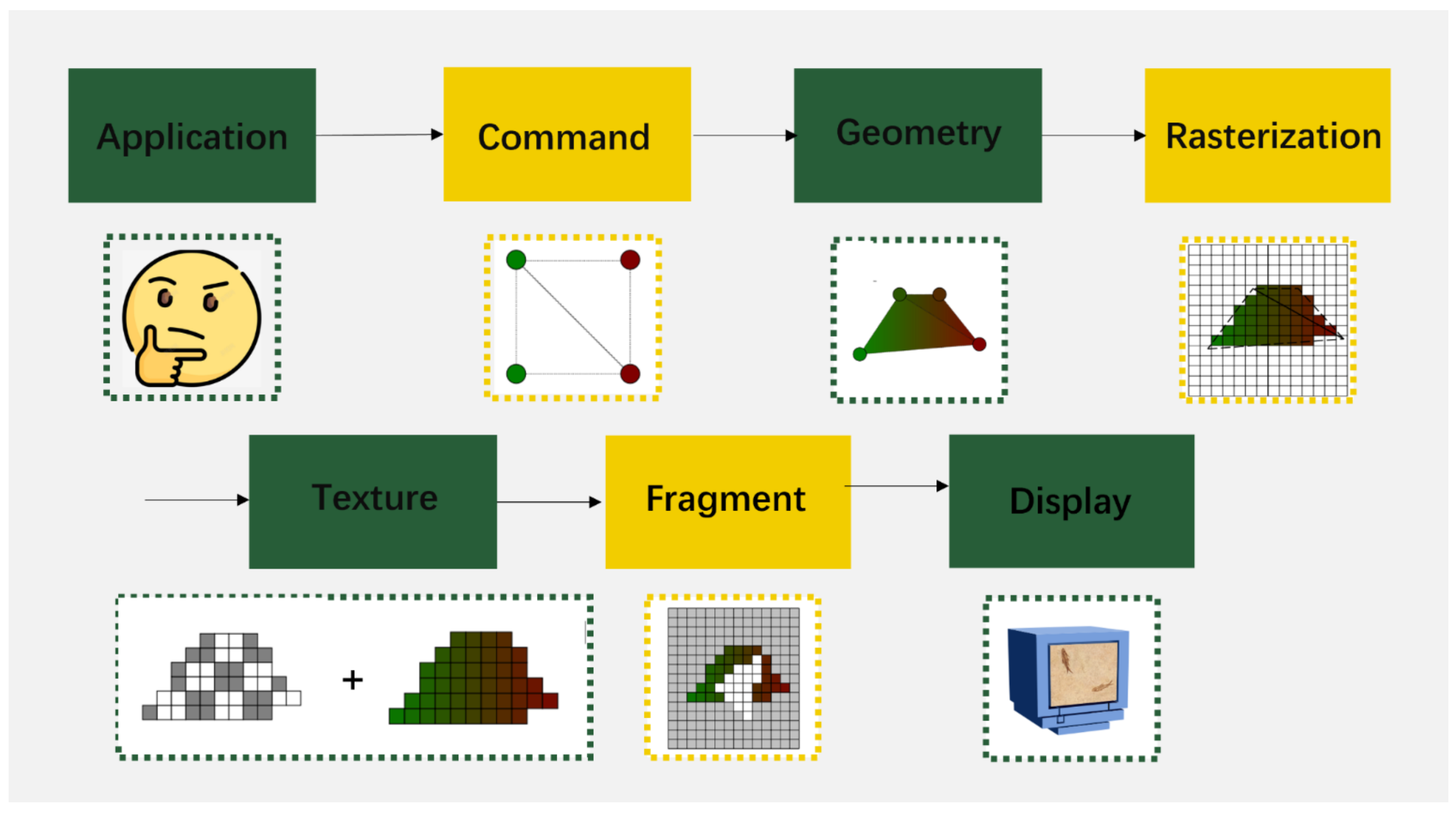


Fig. 3: Pipeline of classic renderer.

Rasterization: In this stage, we sample the triangle to generate fragments and interpolate colors and coordinates.

Texture: We perform texture transformation and projection, calculate texture addresses, and apply texture filtering to enhance image quality at this stage.

Fragment: We do texture application, various tests and blending techniques to determine the final color of each pixel.

Display: Everything shows up like magic in this stage!

Future Plan

Advancing from classic rendering to neural rendering.

Neural rendering is an advanced paradigm in rendering technology, integrating sophisticated differentiable operators to leverage advanced AI techniques for enhanced rendering quality and efficiency. While most neural rendering methods remain largely theoretical, our goal is to bridge the gap between research and practical application by incorporating these advanced features into our renderer, thereby enabling users to experience cutting-edge quality effortlessly.