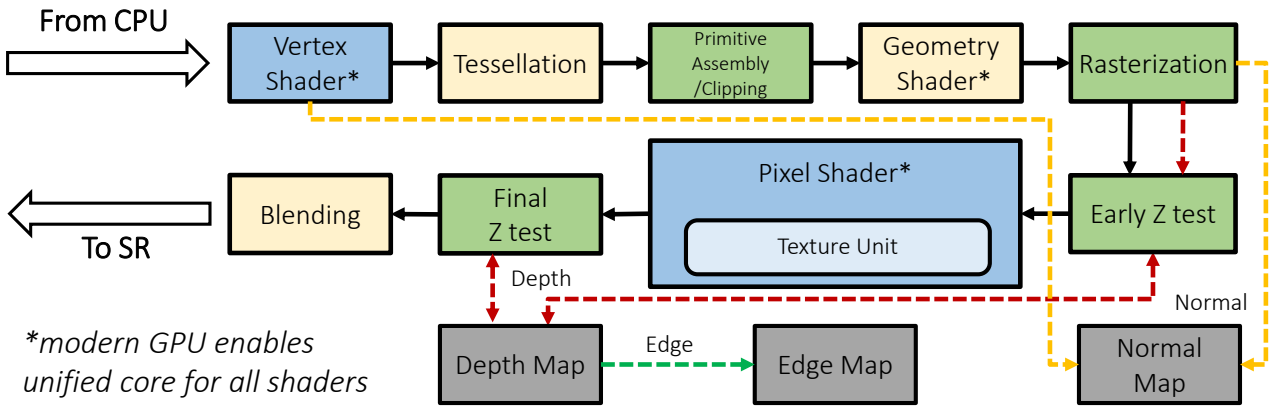

Figure 1: Standard GPU rendering pipeline.

Appendix/Supplemental material

Rendering Pipeline Analysis

In real-time computer graphics, the rendering pipeline describes the process of generating a 2D frame from 3D models. It is essential to break down the pipeline and conduct a detailed analysis to understand how data is processed within the pipeline and identify key extractable information.

Fig. 1 reflects the standard rendering pipeline, which consists of multiple stages. Typically, the CPU prepares all the required data—such as vertex positions, textures, and other rendering-related information—and sends it to the GPU for processing. Once the data is received, the GPU initiates the rendering process as follows:

Vertex Processing: The GPU begins by processing vertices in the vertex shader. This stage involves transforming vertex coordinates from object space to screen space, as well as applying lighting and other per-vertex operations.

Tessellation (Optional): After vertex processing, the tessellation stage subdivides the original mesh into finer triangles. This stage includes a tessellation control shader, which determines how much tessellation should be applied, and a tessellation evaluation shader, which computes the positions of the new vertices.

Primitive Assembly & Clipping: Following the tessellation, vertices are assembled into primitives (e.g., triangles). During this stage, Clipping is performed to discard or modify primitives that are partially or completely outside the view frustum, ensuring that only visible portions of the geometry are processed further.

Geometry Shader (Optional): A geometry shader can further modify these primitives or generate new ones.

Rasterization: The triangles are then rasterized, converting them into fragments (potential pixels) that correspond to specific screen locations covered by the triangles.

Early Z Test: Before further fragment processing, an early Z (depth) test is performed to discard fragments that are occluded by closer geometry. This test uses depth information from the depth buffer to determine which fragments are visible.

Pixel Shading: The visible fragments that pass the early Z test are processed in the pixel shader to determine their final color. During this stage, textures and other effects are applied. Texture units fetch the relevant texture data from the texture cache, sample texels based on the fragment's coordinates, and filter these samples to compute the final color.

Final Z Test: After fragment shading, a final depth test is conducted, where the depth of the shaded fragments is compared against the current depth buffer values to confirm their visibility. This test ensures that only fragments with the correct depth values are written to the frame buffer. Additionally, if template (stencil) testing is used, it is also applied at this stage to enforce more complex visibility rules.

Blending (Optional): Blending is used to combine the color of new fragments with the color already present in the frame buffer to achieve effects such as transparency and translucency. During blending, the source color is combined with the destination color based on blending functions and equations specified by the developer.

Once the blending is complete, the visible and shaded fragments are stored in the frame buffer, resulting in the final rendered frame, which is now ready for display. In the motivation section of the main text, we identified the relevant rendering information by analyzing the SR error map and feature visualization. By examining the rendering pipeline, we can further determine at which stage the specific information first appears and when the complete map becomes available.

For depth information, the depth value for each fragment begins to be calculated during rasterization, producing preliminary depth data. The incomplete depth map becomes accessible as early as the early Z test, which helps to discard fragments occluded by closer geometry. The final depth map is updated after the final Z test, ensuring that only visible fragments are written to the frame buffer. Since object edges are derived from the depth map, their availability aligns with the processing timeline of

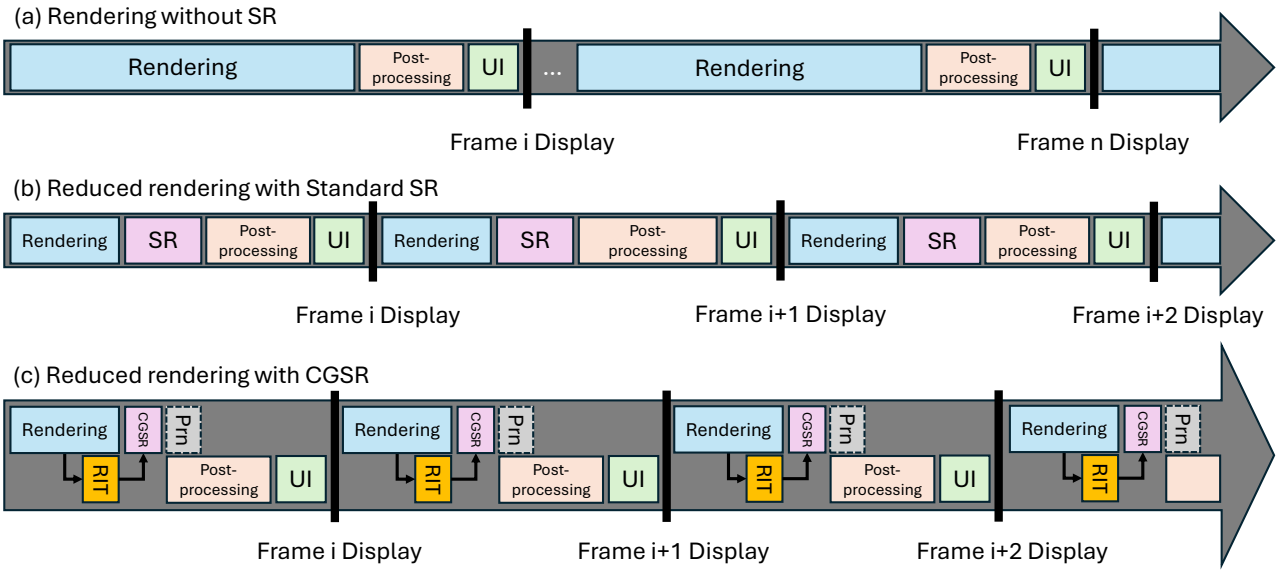

Figure 2: Comparison of rendering and super-resolution timelines.

the depth information. For normal vectors, these are computed during the vertex processing stage and then interpolated across the surface of the triangles during rasterization. As a result, the normals for each fragment are determined by the end of the rasterization process. Therefore, the final normal vector map is ready nearly simultaneously with the depth information.

Rendering and Super-Resolution Timeline

Based on the previous analysis, it is evident that all selected rendering information is generated earlier than the final rendered frame. This enables the pre-processing of this information and masks latency in the pipeline, thereby avoiding additional delays in the Super-Resolution (SR) process. To illustrate the advantages of the CGSR design, we provide an overview of the rendering workflow and our CGSR SR workflow in Fig. 2.

The top line of the figure represents the traditional rendering workflow, including optional post-processing effects and UI processes. The middle line depicts the workflow for reduced rendering combined with standard SR. By applying SR, the original rendering resolution can be reduced—typically by a factor of 1/2 to 1/4—thereby decreasing computational workload while restoring final image quality later. This approach significantly reduces the overall frame time and enhances the user experience.

The bottom line shows the reduced rendering combined with our CGSR-optimized SR process. In this workflow, a lightweight information transform block is integrated along the rendering pipeline, significantly reducing the SR process's computational overhead. The yellow box labeled RIT represents Rendering Information Transform. This process starts running before the rendering is completed and operates in parallel. The gray dashed box in the SR workload highlights the potential for optimization through our rendering-aware dynamic pruning, with Prn indicating the pruning process. As the optimization relies on the temporal content, the portion that can be optimized varies over time.

Dataset Generation and Representation

As outlined in the experiment section of the main text, we provide an overview of our dataset generation process. We enhanced the AttilaSim GPU simulator to generate additional rendering information alongside the rendered frames. The dataset includes three real-world game traces supported by the simulator. Although these games are not the latest, they offer valuable test cases for SR applications. SR techniques are particularly effective in upscaling images, videos, and games from lower resolutions (e.g., HD) to higher resolutions (e.g., 4K), thus significantly improving the visual experience. Additionally, while the dataset includes traces provided by the simulator developer, it is also possible to capture new game traces, provided they use a compatible graphics API, which offers flexibility for future expansions.

For training, validation, and test sets, we selected three types of rendering information and chose to represent them in an image format, consistent with the rendered frame. Specifically, each RGB channel corresponds to a distinct type of rendering information: R-channel for object edges, G-channel for depth, and B-channel for normal vectors. This representation ensures that the rendering information is aligned with the rendered frame, facilitating effective learning by the network. For depth information, the values are normalized to a range of [0, 1]. For normal vectors, we map the camera-facing direction (0°) to 0

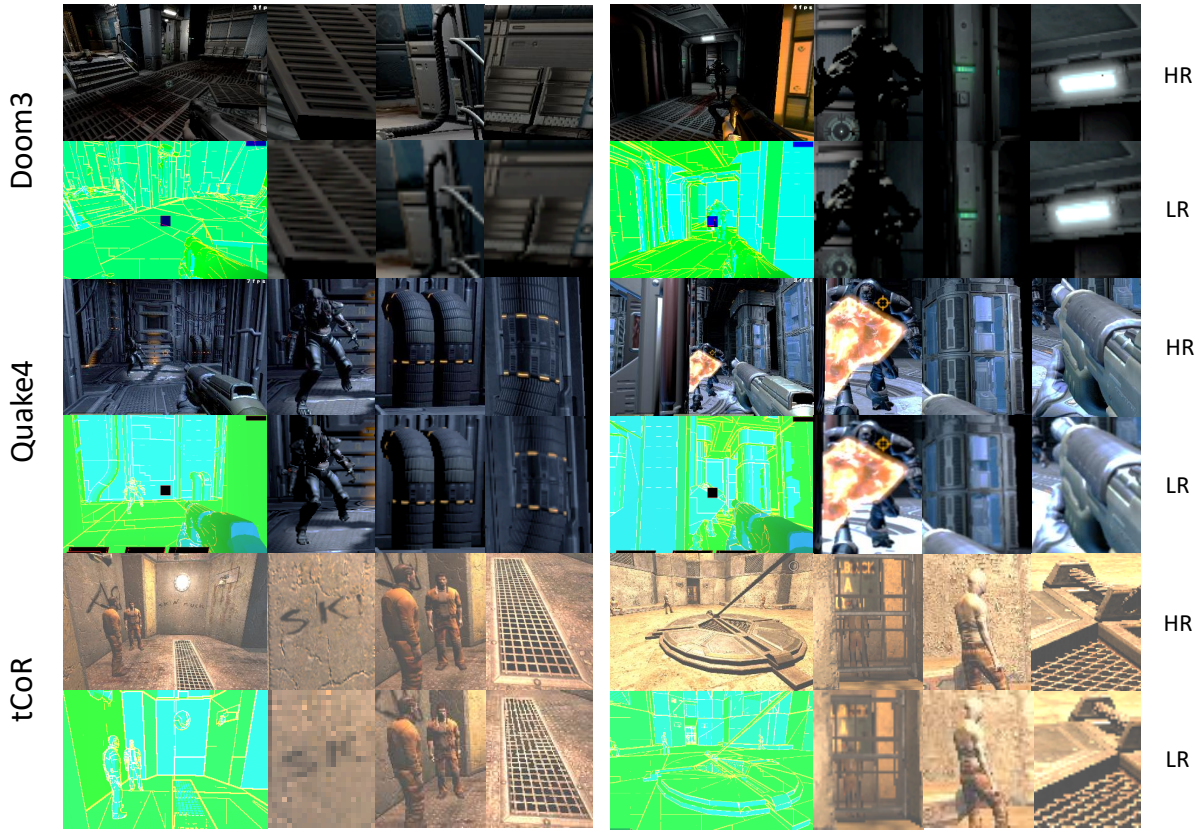

Figure 3: Additional dataset visualizations across different traces.

and the side-facing direction (90°) to 1, and then normalize the values accordingly. Object edges are represented using a boolean value, where a pixel is marked as an edge pixel if it meets the criteria.

Fig. 3 illustrates three different examples from the dataset of the selected game traces. Both low and high-resolution frames are rendered under identical graphics settings, with corresponding rendering information, ensuring consistency and accuracy. The boxed areas within the rendering information map are attributed to UI elements or lighting maps. UI elements are typically treated as having a depth of 0 and a normal vector orientation of 0°, while lighting maps, which apply textures for lighting effects, can also result in normal vectors of 0 and obscure the original objects. These practices are standard in game development. In practice, UI elements typically demand less computational power and can be rendered directly at high resolution, thus often bypassing the need for SR. Additionally, these areas only cover a portion of the scene, leaving other regions that still provide meaningful information for SR.