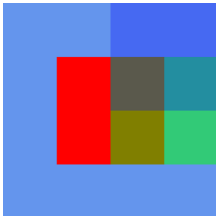


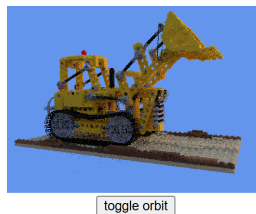
Worksheet 18: Blending, Gaussian splatting, and render pipeline

Reading	RTR: 6.6, 13.9 [Kerbl et al. 2023, Kheradmand et al. 2025]																
Purpose	This worksheet introduces the popular concept of 3D Gaussian splatting for image-based rendering [Kerbl et al. 2023]. This type of splatting requires projection of a point cloud to screen space and depth-sorted blending of Gaussian functions associated with these points. As an alternative to depth sorting, we consider stochastic rejection of splats [Kheradmand et al. 2025] and progressive updates when the camera is kept still.																
Part 1 Blending of 2D point splats 	We splat points by letting the vertex shader build a screen-aligned quad for each point. Thus, we need a triangle strip of 4 vertices for each point, but we also want to splat many points with the same draw call, so instanced drawing should be used. - Set up your vertex buffer to use instance step mode, so you only need one position and one color for each point. - Use instancing to draw three points of width 0.5 colored red, green, and blue (start from Part 3 of Worksheet 1, for example). - Set the alpha-value of the colors to 0.3 and blend the points using conventional alpha blending.																
Part 2 Gaussian splatting 	Instead of defining the splats yourself, load them from <code>lego-2d-sorted.ply</code> using the function <code>readPlyParser</code> in the library file <code>PLYParser.js</code> . The function returns an array of objects. Use the first one. The object has a binary data field that you can write into your vertex buffer. The layout of the attributes associated with this vertex buffer is as illustrated in the table. <table><tr><th>Position</th><th>Half side lengths</th><th>Direction 1</th><th>Direction 2</th><th>Color</th><th>Opacity</th><th>Σ_{ss}^{-1} (xx, xy, yy)</th><th>Depth</th></tr><tr><td>vec2f</td><td>vec2f</td><td>vec2f</td><td>vec2f</td><td>vec3f</td><td>f32</td><td>vec3f</td><td>f32</td></tr></table> The stride between each vertex/splat in the buffer is <code>sizeof['mat4']</code>. Use offsets to separate the vertex data into input arguments for the vertex shader. Note that the splats are in screen space and the depth is only used in Part 3. - Change your vertex buffer to use the loaded splats and make the resolution of your canvas available as a uniform variable. - Use the position, half side lengths, and directions in the vertex shader to splat the points. Convert the shifted quad vertex position to NDC before returning it from the vertex shader and pass the unmodified screen space splat position to the fragment shader together with color, opacity, and the upper triangle of the inverse covariance matrix. - Compute the alpha value (α) for the splat using the xy-part of the built-in screen space fragment position retrieved as an input argument of the fragment shader (<code>@builtin(position) fragCoord: vec4f</code>) and inserted into a Gaussian kernel with shape defined by the inverse covariance matrix, mean given by the unmodified splat position, and amplitude given by the opacity.	Position	Half side lengths	Direction 1	Direction 2	Color	Opacity	Σ_{ss}^{-1} (xx, xy, yy)	Depth	vec2f	vec2f	vec2f	vec2f	vec3f	f32	vec3f	f32
Position	Half side lengths	Direction 1	Direction 2	Color	Opacity	Σ_{ss}^{-1} (xx, xy, yy)	Depth										
vec2f	vec2f	vec2f	vec2f	vec3f	f32	vec3f	f32										

Worksheet 18: Blending, Gaussian splatting, and render pipeline**Part 3**
Stochastic Gaussian splatting

In general, we cannot assume that our splats are sorted correctly for an arbitrary view. To work with unsorted splats, load `lego-2d.ply` instead of `lego-2d-sorted.ply`. Instead of blending, we will now use stochastic selection and the depth buffer to sort our splats probabilistically.

- Ensure that the depth is included in the vertex buffer and replace the alpha blending in the render pipeline with a depth buffer. Pass the depth value into the rasterization pipeline by inserting it as the z-coordinate of the vertex position in the vertex shader.
- For the stochastic selection, we need to seed a pseudo-random number generator. Use the screen space coordinates (`fragcoord.xy`) and the instance index as the seed. Retrieve the instance index as a built-in input argument of the vertex shader and pass it to the fragment shader using flat interpolation (`@interpolate(flat)`).
- In the fragment shader, generate a random number $\xi \in [0,1]$ and discard the fragment (call `discard`;) if $\xi > \alpha$. Otherwise, return the splat color with the alpha channel set to 1.

Part 4
Projection to screen space

To dynamically change our view of the Gaussian splats, we need to project the original 3D Gaussians to the 2D screen space splats that we have been using in the previous parts. Load `lego-simplified.ply` instead of `lego-2d.ply`. The data in this file has a different layout for the vertex attributes:

Position (world space)	Padding (ignore)	Color	Opacity	Σ (part 1) (xx, xy, xz, yy)	Σ (part 2) (yz, zz)	Padding (ignore)
vec3f	u32	vec3f	f32	vec4f	vec2f	vec2u

Your task is now to change the vertex buffer layout according to these world space attributes of the 3D splats and update the vertex shader to compute the screen space information you had before for a given view and camera. To this end, make the projection and view matrices available in the vertex shader by including them in the buffer for uniform variables. Add a button for switching orbiting of the camera around the scene on/off.

Part 5 (optional)
Progressive updating

☒ Progressive Updates
toggle orbit

The stochastic Gaussian splatting is noisy. We can fix this problem for a static camera using progressive updating. Maintain a frame number that is incremented when the camera is static and zero otherwise. Upload the frame number as a uniform variable and use it as another seed input for the pseudo-random number generator. Create an off-screen texture for storing the stochastic splatting result. Then create two additional textures for a ping-pong rendering scheme and write a separate set of shaders for a progressive updating pass after the splatting. Set up one texture to be a render target and a copy source and one to be a texture binding and a copy destination. Give the fragment shader for the progressive updating access to the textures holding the new splatting result and the previous averaged result and combine them into an average of all the frames since the camera was last modified. Create a checkbox for switching progressive updating on/off.