LayerPaint: A Multi-Layer Interactive 3D Painting Interface

Chi-Wing Fu **Jiazhi Xia** **Ying He**
School of Computer Engineering, Nanyang Technological University, Singapore
cwfu@ntu.edu.sg xiaj0002@ntu.edu.sg yhe@ntu.edu.sg

ABSTRACT
Painting on 3D surfaces is an important operation in computer graphics, virtual reality, and computer aided design. The painting styles in existing WYSIWYG systems can be awkward, due to the difficulty in rotating or aligning an object for proper viewing during the painting. This paper proposes a multi-layer approach to building a practical, robust, and novel WYSIWYG interface for efficient painting on 3D models. The paintable area is not limited to the front-most visible surface on the screen as in conventional WYSIWYG interfaces. We can efficiently and interactively draw long strokes across different depth layers, and unveil occluded regions that one would like to see or paint on. In addition, since the painting is now depth-sensitive, we can avoid various potential painting artifacts and limitations in the conventional painting interfaces. This multi-layer approach brings in several novel painting operations that contribute to a more compelling WYSIWYG 3D painting interface; this is particular useful when dealing with complicated objects with occluded parts and objects that cannot be easily parameterized. We evaluated our system with 23 users, including both artists and novice painters, and obtained positive experimental results and feedback from them. The user study results demonstrate the efficacy of our novel interface over conventional painting interfaces.

Author Keywords
3D painting, WYSIWYG interface, depth segmentation

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: User interfaces; I.3.4 Graphics Utilities: Paint systems

General Terms
Algorithms, Design.

INTRODUCTION

Painting 3D models is an important operation in computer graphics, virtual reality, and computer-aided design, as well as computer entertainment and gaming. A desired 3D painting system should allow the user to paint on the object surfaces in an efficient and intuitive way. Certain 3D input devices, such as haptics, provide the users with high degree of spatial freedom to directly control the brush movement in the 3D space of the object. However, the cost of the hardware usually limits its applications to experts, while many artists still prefer to paint 3D models with conventional 2D interfaces, such as the tablet and mouse, as demonstrated in traditional painting with 2D drawing canvas.

To maximize 3D painting capability with 2D input devices, Hanrahan and Haeberli pioneered the WYSIWYG design [12] that allows the users to directly paint with different types of pigments and materials onto the 3D surfaces by projecting the footprint of the brush from the screen space to the texture space. There are two widely-used techniques, screen- and tangent-space, to define the projection. The former projects the brush footprint onto the surface using inverse viewing transformation whereas the latter places the footprint in the tangential surface at the brush position and projects the footprint onto the surface in the direction parallel to the surface normal. The WYSIWYG approach is highly intuitive, easy-to-implement, and it usually leads to expectable results that are faithful to what we have seen on the screen. However, the painting style of all the existing WYSIWYG systems can be awkward, due to the difficulty in rotating the object for proper viewing during the painting, and such a situation can be even worse when the 3D objects have complex topology and geometry, for example, those with occluded region(s) that cannot be adequately revealed from many different viewing directions.

This paper presents *LayerPaint*, a novel WYSIWYG painting system that only requires low-cost 2D input devices such as the tablet and mouse. In contrast to the existing WYSIWYG systems, *LayerPaint* uniquely allows the users to efficiently paint long strokes over multiple layers without worrying about the occlusions. Our design is built upon a series of multi-layer methods that run at interactive speed with the help of the GPU on the graphics hardware, so that we can support interactive painting operations over multiple depth layers. Once the artist picks a certain view of an input 3D model, our interface first carries out multi-layer segmentation to partition the depth-peeled layers into connectable regions and build pixel-level and region-level connectivity information at interactive speed. Hence, we can design and implement a layer-aware painting mechanism that is sensitive to both region occlusion and region boundary, while support-

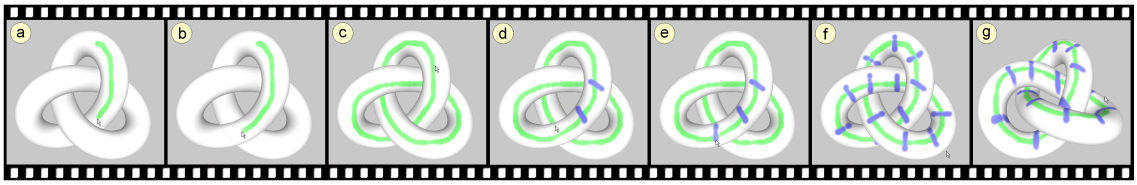

Figure 1. Our multi-layer approach brings in novel painting operations that run at interactive speed: given a 3D model, we can draw a long stroke on it, see (a)-(c), and when the stroke gets occluded (see (b) and note the cursor), the hidden region can popup automatically. We can draw the entire green line, see (c), in only a single stroke. Users can also selectively pop-up any hidden region with a mouse click, see (d), and draw on this pop-up region, see (e). It takes only 35 seconds (with a mouse) to complete the drawing shown in (f), see (g) for another view.

ing the drawing of depth-sensitive long strokes across different layers. Moreover, to further take advantage of the available multi-layer connectivity information, we design also a collection of multi-layer painting operations, for example, an interactive region select-and-hide mechanism that one could automatically unveil occluded regions while painting or intentionally unveil the selected regions with mouse/tablet clicks.

As demonstrated in Figure 1, the users can draw a very long stroke that spans not only the front-most visible layer but also the hidden layers. When the stroke enters a hidden layer, *LayerPaint* can automatically pop up the hidden region for the users to continue the stroke. The users can draw continuous and smooth strokes without changing the viewpoint. This feature is highly desired, especially for painting on models with highly complex occlusions. Powered by GPU-based layer segmentation, interactive painting with this feature can be supported, see also the implementation and result section. The major contributions of this paper are summarized below:

- First, this paper proposes a multi-layer approach to build a compelling WYSIWYG painting interface for 3D models. This is the first paper (we noticed) that explores the use of multi-layer information for interactive 3D painting.
- Second, we propose a series of novel multi-layer methods, including the GPU-based multi-layer segmentation to partition depth-peeled layers into regions with pixel-level and region-level connectivity information; layer-aware painting algorithm to facilitate the drawing of depth-sensitive strokes automatically over suitable depth layers, see Figure 1; and interactive region popup and rendering algorithms to allow automatic or intentional unveiling of occluded regions. Note that all these algorithms are designed to run at interactive speed in order to support interactive painting over multiple layers.
- Third, several multi-layer operations are introduced into the WYSIWYG painting interface: 1) layer-aware painting; 2) interactive region select and pop-up; 3) interactive paint-to-hide; and 4) layer-aware object rotation. These are novel painting operations available only after we explore interaction methods with multiple depth layers.
- Last, we carefully integrate the above ideas as a working user interface and carry out a user study that further demonstrates the efficacy and applicability of *LayerPaint*.

PREVIOUS WORK

3D Navigation. Painting on real 3D models involves a high degree of freedom in the spatial movement of the paintbrush and also in the placement of the 3D object. Manipulating virtual objects with mouse controls usually takes more time than direct manipulation of real objects [23]. The haptic devices are capable of providing such a high degree-of-freedom while providing also the sense of touch with force feedback to the users. Usually, this kind of devices is armed with a pen-like probe, which can be manipulated like the pen (the painting brush) in the 3D painting. These unique features make haptic devices highly desirable in narrowing the gap between real-world 3D painting and virtual 3D painting [5, 21, 10]. Balakrishnan and Kurtenbach [3] proposed using non-dominant hand to operate the virtual camera controls typically found in 3D graphics applications, thus freeing the dominant hand to perform other manipulative tasks in the 3D scene. Khan et al. [13] proposed a surface-based camera control technique that allows the user to focus on the task at hand instead of continuously managing the camera position and orientation. McCrae et al. [17] proposed a multi-scale system that allows consistent 3D navigation at various scales, as well as real-time collision detection without precomputation or prior knowledge of the geometric structure.

Painting devices. Several innovative painting devices have been developed. Among these, we notice the work of Schkolne et al. [20], who proposed *Surface Drawing* for users to construct 3D shapes with various hand and tangible tools in a semi-immersive virtual environment, the work of Ryokai et al. [18, 19], who proposed the *I/O Brush* to explore the combination of colors, textures, and movements in everyday materials, and the work of Vandoren et al. [22], who created a natural painting interface on an interactive table using a brush with infrared light emitting fibers. The haptic feedback and an accurate brush footprint are added to the IR-brush to enable the contact area tracking.

3D painting/modeling interface. A considerable amount of research effort have been devoted to explore 3D paintings with conventional 2D input devices. Hanrahan and Haeberli pioneered the WYSIWYG system for the users to directly paint on 3D surfaces by mapping the brush footprint to texture [12]. Agrawala et al. [1] presented an intuitive interface for painting on scanned surfaces using a physical object as a guide. By moving the sensor of the space tracker over the surface of a physical object, user-picked colors can be painted onto the corresponding locations on the scanned mesh. Grimm and Kowalski [11] presented a painting interface to control what the object should look like under different lighting and viewing conditions. The system can render the object under novel lighting conditions and a new viewing angle by combining various painting controls. DeBry et al. [7] presented an octree-based paint and texture mapping system that completely avoids the surface parameterization.

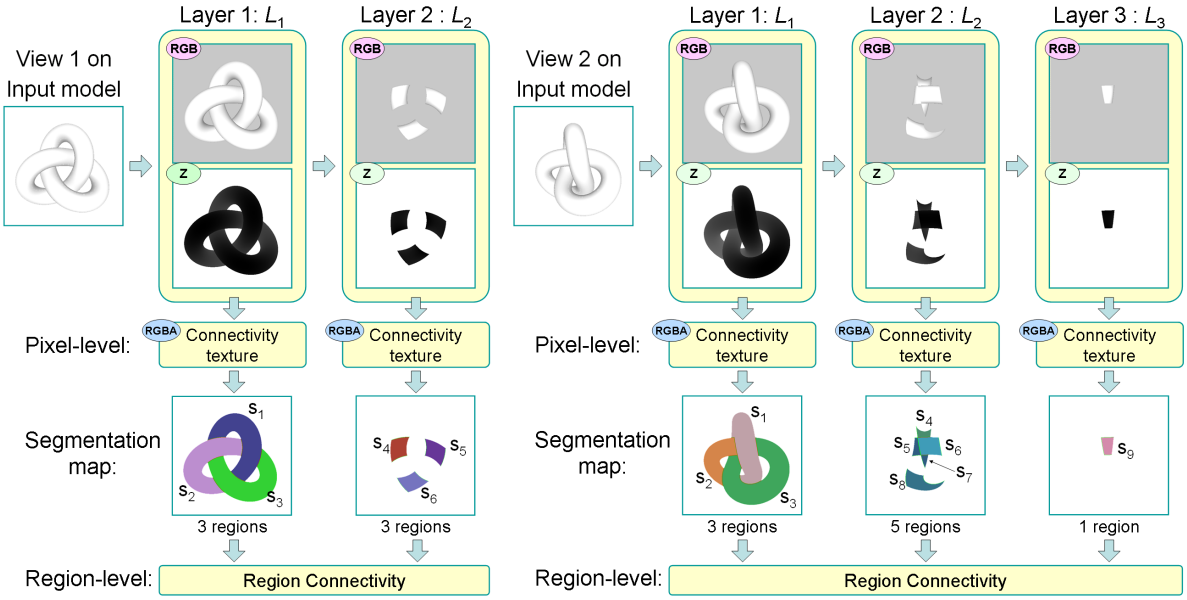

Figure 2. Multi-layer segmentation on two different views of the Trefoil Knot model: depth peeling is first applied to generate one set of color and depth textures per layer; segmentation is then further applied to produce pixel-level connectivity, segmentation maps, and region-level connectivity.

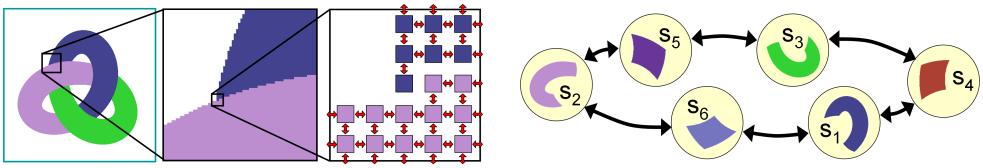

Figure 3. Pixel-level (left) and region-level (right) connectivity: we illustrate these two levels of connectivity using view 1 shown on the left hand side of Figure 2; note that we zoom into a region on layer L_1 (of view 1) to illustrate the pixel-level connectivity.

Due to the adaptive nature of this approach, detail is created on the map only, as required by the texture painter. Bae et al. [2] proposed a 3D curve sketching system for professional designers to iterate directly on concept 3D curve models.

Cutaway visualization/illustration of 3D models. Burns and Finkelstein [6] developed a system for authoring and viewing interactive cutaway illustrations of complex 3D models. Li et al. [15] presented a method for generating cutaway renderings of polygonal scenes at interactive frame rates, using illustrative and non-photorealistic rendering cues to expose objects of interest in the context of surrounding objects. More recently, Li et al. [14] presented a system for generating interactive exploded views for complex 3D models.

The proposed *LayerPaint* interface differs from conventional WYSIWYG systems. Taking advantage of the GPU computational power, *LayerPaint* can efficiently allow us to paint not just on the front-most visible area, but also on any underlying regions. Since it partitions depth-peeled layers of the current view into connectable regions, the automatic region popup and rendering algorithms can provide automatic or intentional reveal of any occluded object part. With these unique features, the users can draw very long strokes over multiple layers without needing to change the viewpoint or worrying about the occlusions.

MULTI-LAYER SEGMENTATION

Multi-layer segmentation is a view-dependent pre-processing step, aiming at supporting the painting operations to be presented in the next section. Basically, its goal is to generate multi-layer segmentation and connectivity information, including the pixel-level connectivity, the segmentation maps, and the region-level connectivity. Since it has to be performed every time after the user changes the object view, we aim at a highly efficient multi-layer segmentation that can run interactively with the help of the GPU.

Depth Peeling. Figure 2 illustrates the entire multi-layer segmentation process with two example views of the Trefoil Knot model. First, we apply the conventional depth peeling method [9] to the user-selected view on the 3D model with backface culling enabled. Note that this process can be performed entirely on the GPU, and can be further accelerated with recent methods [4, 16]. To facilitate our discussion in the paper, we denote L_1 as the front-most depth layer (layer ID = 1), L_2 as the second layer (layer ID = 2), etc. Concerning the depth-peeled layers, we have the following property:

> *For any foreground (non-background) pixel $p(x, y)$ in L_j ($j > 1$), there must be a foreground pixel at (x, y) on L_i (for $i = 1$ to $j - 1$) with a smaller depth value than that of p on L_j.*

Such a property can be highly useful for accelerating the multi-layer rendering to be presented in the "Multi-layer Operations" Section. And the result of this step is one set of color and depth images (GPU textures) for each depth layer, see the first row of Figure 2. Note that occlusion query on GPU is used to count foreground pixels per layer.

Connectivity and Segmentation Information. Our second step to build the connectivity and segmentation information contains the construction of 1) pixel-level connectivity; 2) segmentation map; and 3) region-level connectivity. Since these operations are heavily related to the GPU, please refer to the Appendix for the detail.

Table 1. Comparing *LayerPaint* and *Eisemann et al.* [8]

	LayerPaint	Eisemann et al. [8]
Goal	3D painting	Vector graph creation
Algorithm	depth peeling(GPU)	ray casting(CPU)
Requiring region level connectivity	yes	yes
Requiring pixel level connectivity	yes	no
Layer Boundary	visibility	graph conflict & geodesic
Performance	real-time	offline

In fact, there are other ways to segment depth-peeled layers. For instance, a recent work on creating vector graphs from 3D models [8] explores also multi-layer segmentation. However, our goal is quite different from this work (See Table 1 for a detailed comparison with Eisemann et al. [8]), where we aim at high performance and robust segmentation (and the production of multi-layer connectivity information) with the GPU, so that we can support the proposed painting operations at interactive speed.

MULTI-LAYER OPERATIONS

After multi-layer segmentation, we could perform various interactive operations over multiple layers. The following multi-layer operations/interactions are proposed:

(1) Layer-aware painting

Unlike previous WYSIWYG screen painting interface, layer-aware painting supports the drawing of long strokes in a depth-sensitive manner, see also Figure 1. In detail, we maintain two screen-sized maps storing paintable and trackable information during program runtime:

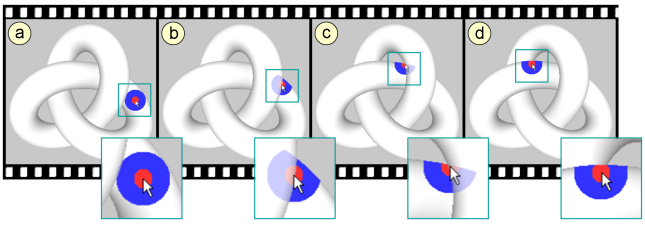

Figure 4. Paintable (red) and trackable (include both red and blue) regions over multiple layers as we draw a stroke on the screen.

Paintable Region: When the user starts a paint action, say with a touch pen or a mouse, we first obtain the on-screen pixel location, say (x_0, y_0), and lookup the layer ID, say L_i, corresponding to the pixel currently visible on the screen. Note that with the region popup and hiding operations to be presented later, the visible pixel at (x_0, y_0) may correspond to an underlying layer. Then, starting from (x_0, y_0, L_i), we can apply breath-first search with the pixel-level connectivity information to recursively visit all neighboring pixels within a user-defined brush radius, say R. Since the breath-first search could visit the same pixel location more than once (note that it may go over another layer and loop back), we use a screen-sized map, say $M_p(x, y)$, to bookmark the layer ID for each visited (paintable) pixels. Hence, we can ensure that only one certain layer is paintable for each screen pixel location. And since we use breath-first search, the pixel nearer to (x_0, y_0, L_i) is guaranteed to be picked first. As a result, $M_p(x, y)$ can tell us the paintable region around (x_0, y_0, L_i). Figure 4 shows the paintable regions in red, and as the cursor moves, the paintable region may fall (completely or partially) behind other layer(s) as demonstrated in (b) and (c), and the paintable region is also sensitive to the region border, see (d).

Trackable Region: To support the drawing of long strokes with mouse/touch-pen drag while maintaining depth continuity, we need to determine the layer ID for the succeeding mouse cursor location, say (x_1, y_1), after the cursor just moves away from (x_0, y_0). Generally, we need to check all multi-layer (foreground) pixels existed on (x_1, y_1), and find out the one that is the nearest to (x_0, y_0, L_i) against the pixel-level connectivity information, which works like a graph data structure in this case. Since tracking from multi-layer pixels at (x_1, y_1), especially from those on unmatched layers, could be computationally expensive, we take an alternative approach to compute a trackable region, say M_t, from (x_0, y_0). Its advantage is that we can build this trackable region efficiently by re-using the information available in the paintable map, i.e., the layer ID. In particular, rather than stopping the breath-first search at R (when constructing the paintable region), we can continue the search until a much larger radius, say R_{max}. Thus, we can obtain the layer ID for more pixels around (x_0, y_0). Then, such a trackable map can support a fast lookup of the layer ID when the cursor just moves to (x_1, y_1). Furthermore, we can also take advantage of the layer ID coherence to avoid rebuilding of the entire trackable region on successive cursor movements. See also Figure 4 for the trackable region, which includes both the red and blue areas.

The Painting Algorithm: Given the paintable and trackable regions, our painting algorithm works as follows:

On-Mouse-Down (x_0 , y_0 , object) {
  $L_i \leftarrow$ lookup layer ID of visible pixel on (x_0 , y_0)
  (M_p , M_t) $\leftarrow$ build_maps(x_0 , y_0 , L_i)
  paint_object(object , M_p)
}
On-Mouse-Drag (x_1 , y_1 , object) {
  $L_i \leftarrow$ lookup layer ID from M_t
  $M_p \leftarrow$ build_paintable_map(x_1 , y_1 , L_i)
  $M_t \leftarrow$ update_trackable_map(x_1 , y_1 , L_i , M_t)
  paint_object(object , M_p)
}

In practice, before we lookup the layer ID L_i, we need to check if the cursor moves to a non-foreground pixel or a non-depth-connected region, i.e., outside M_t. If this happens, we have to ignore the mouse action to avoid mis-painting.

Avoiding Color Bleeding: One potential problem with WYSIWYG painting is accidental painting on irrelevant layers. This weird case may also happen in *LayerPaint*, see the middle column of Figure 6, where the cursor locates very close to the end of a suggestive contour line, and so, the breath-first search could go around the end of the contour line and label also the pixels on the opposite side of the contour.

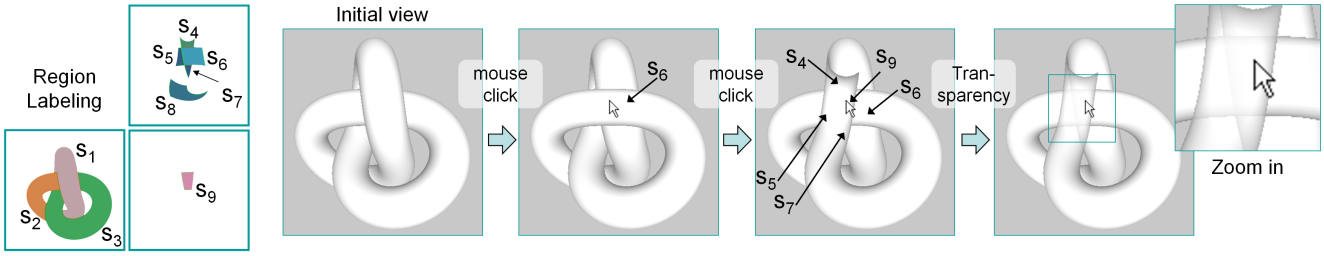

Figure 5. Region sorting and rendering after intentional region select and popup.

To resolve this problem, we put an additional constraint in the breath-first search while building the paintable map. Here we keep track of the distance the breath-first search moved so far from the starting cursor location, i.e., (x_0, y_0). If the distance goes above $\sqrt{2}R$ at a certain pixel being visited, we stop the search there even though the pixel is still within the brush radius R, see the fixed result in the last column of Figure 6. Note that the second row shows another views that better reveal how unpleasing color bleeding could be.

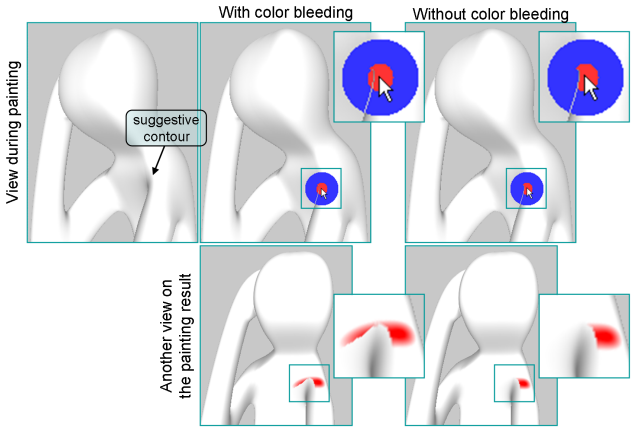

Figure 6. With (middle) and without (right) Color Bleeding.

(2) Interactive Region Select and Popup

The multi-layer segmentation and connectivity information facilitates two kinds of interactive region select and popup:

1. *Intentional region popup*: The user can click on the screen, say (x_2, y_2), to unhide the region below the currently visible pixel. First, our interface can look up the layer ID, say L_i, of the currently visible pixel at (x_2, y_2). Then, it checks the layer behind L_i to see if there is any foreground pixel just below the visible pixel. If it is the case, it will lookup the region ID, say s_i, of the region containing (x_2, y_2, l_2) from the segmentation map and do a region popup on s_i. Otherwise, it looks up the segmented region containing (x_2, y_2, L_1), i.e., the front-most layer, and do a region popup with this front-most region. Thus, the user can also cycle through different available regions on (x_2, y_2), see (c) and (d) in Figure 1 for an example of intentional region popup.

2. *Automatic region popup*: After each cursor movement in layer-aware painting, if the painting location, say (x_3, y_3) with layer ID L_i, is not the currently visible pixel on the top, our system looks up the segmented region containing (x_3, y_3, L_i) and do a region popup on the region, see (a) and (b) in Figure 1 for an example. Note that the user can disable this feature if they want to keep the on-screen region visibility ordering during the painting.

Region popup and sorting: Every time after we build the connectivity and segmentation information, we keep a front-to-back sorted list of segmented region IDs, for example, for view 2 shown in Figure 2 (right) (also Figure 5), we maintain a list of nine region IDs: $s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_9$, where initially, the three segmented regions on the first layer, i.e., s_1 to s_3, are in the front, and so on. If s_i is the region to be popup (either in an intentional or automatic region popup), s_i will be moved to the front of this sorted list.

In case it is an intentional popup, we popup also the regions connected to s_i (and potentially the regions further connected to the first region group in a recursive manner). In practice, we do a two-level recursion, so that we can better reveal the surroundings around s_i. Note that for those additional regions popup with s_i, we sort them according to their layer IDs and move them to the front of the sorted list from back to front, e.g., if the initial sorting order is

$$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow s_8 \rightarrow s_9 \ ,$$

and s_4 (on 2nd layer) is intentionally selected for popup with s_1 (on 1st layer) and s_9 (on 3rd layer) connected to it (see Figure 5 for the region labels), s_1 and s_9 will first be moved to the front of the list after s_4:

$$s_4 \rightarrow s_1 \rightarrow s_9 \rightarrow s_2 \rightarrow s_3 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow s_8 \ .$$

Other than region rendering, the sorted list can also affect the layer orderings on screen when we lookup the currently visible pixel, e.g., starting a new drawing stroke from a pixel or doing an intentional region popup.

Back-to-front region rendering: Rendering the segmented regions from back to front can be done by using the fragment program on the GPU. In our current implementation, we do a back-to-front rendering according to the sorted list, but as a speedup, we can first analyze the region sorted list. By considering the depth peeling property stated earlier, when we render a region located in layer L_i $(i > 1)$, say s,

> we can ignore the rendering of s if all regions on layer L_j (for some $j < i$) are to be drawn after s.

In this way, we could prune regions from back to front in the sorted list and generate a smaller number of regions in actual rendering. For instance, given the example sorted list shown previously, we can truncate it into:

$$s_4 \rightarrow s_1 \rightarrow s_9 \rightarrow s_2 \rightarrow s_3 \ .$$

Since s_1, s_2, and s_3 form a complete set of regions on the first layer, they can block regions on the right of s_3 in the original sorted list. Figure 5 shows an example sorting and rendering result of applying intentional region select and popup twice on the same screen pixel location. Different underlying regions (s_6 then s_9) below the initially visible pixel can be unhidden and transparency can also be added into the multi-layer rendering. However, it is worthwhile to note that if transparency is enabled, we can only prune regions against the first layer (but not others), since regions in the first layer always stay opaque, while others are not. And in fact, such rendering feature can be further used to interactively create exploded views that commonly used for visualizing the inner parts of 3D models.

(3) Interactive Paint-to-Hide

In addition to unhiding regions underneath, we support another kind of region-selection, namely the *interactive paint-to-hide*. As shown in Figure 7, we can paint on the view of the segmentation map to remove image region(s) from the view. Hence, we can interactively edit the visibility of various part(s) to best suit our painting purpose. Note that when interactive paint-to-hide is in use, we have to ignore the acceleration trick in layer-based rendering because some regions below the removed part(s) may become visible.

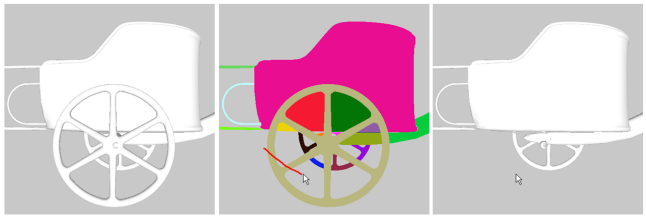

Figure 7. Interactive Paint-to-Hide by region markup.

(4) Layer-aware Object Rotation

When we explore a 3D object with rotation, the rotation center is usually the object center, world center, or a user-picked surface point, where we can locally explore the object while keeping the spatial context around the picked point. With the multi-layer information, we can extend such a rotation to include points over different layers. In our design, when the user clicks on the object surface to initialize a rotation, say (x_4, y_4) on the screen, we pick up the layer ID of the currently visible pixel on (x_4, y_4), and lookup its depth value from the related depth map. Then, we can compute its object coordinate, and rotate about the point. Figure 8 shows a practical usage of this operation. After we popup an underlying region and create an exploded view, we can paint on this underlying region. When we need to further explore its surrounding from different angles, we can pick a surface point in this underlying region, and rotate about it. The advantage of such an operation is that we can keep the spatial context and explore around the picked point even though it is not on the front-most visible layer.

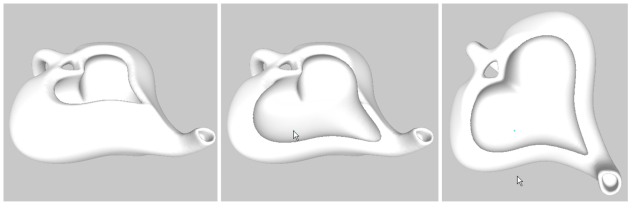

Figure 8. Layer-aware Object Rotation allows us to rotate about surface point on any layer.

IMPLEMENTATION AND RESULTS

We employed several GPU methods in our implementation to support interactive multi-layer segmentation and painting operations. First, we store most data directly on the GPU, e.g., the per-layer textures as frame buffer object (FBO) and the object geometry as vertex buffer object (VBO). Since the color, depth value, and segmentation information are the most-frequently accessed data for all layer-aware operations, we allocate one texture buffer for each of them, as illustrated in Figure 2. Note also that the FBO facilitates efficient texture data read/write with off-screen rendering during the depth peeling process. In addition, since the geometry is static, storing it on the GPU helps to avoid redundant geometry transfer from the main memory via the bus to the GPU.

Figure 9. The setup of *LayerPaint* with the input tablet.

In addition, we experimented with *LayerPaint* on a 64-bit workstation that equipped with a quad-core Xeon 2.50GHz CPU, 8GB memory, and the graphics board GeForce GTX 285 (1GB GPU memory). Figure 9 shows the entire setup including the tablet we used in the experiment and user study. Furthermore, to demonstrate the performance of *LayerPaint*, we measured the time taken for the following four critical procedures in *LayerPaint*:

- *start_stroke* (corresponding to `On-Mouse-Down`) is called every time when the user starts a stroke with a mouse click. It initializes the paintable and trackable maps, and paints the first spot on the object surface;
- *move_stroke* (corresponding to `On-Mouse-Drag`) continues a stroke with a mouse drag. It updates the two maps and paints a new spot on the current mouse location;
- *build_layers* is called whenever the object view changes; this procedure invokes the multi-layer segmentation to construct connectivity and segmentation information;
- *draw_layers* applies fragment shader to render layer by layer using the sorted region list.

Table 2 shows the performance of these procedures as experimented with four different 3D models. Given the timing statistics, we can see that interactive segmentation is achievable with the GPU support and real-time painting of long strokes can also be realized with *LayerPaint*.

Table 2. Performance of *LayerPaint*: Average time taken to perform these operations on four different models (in milliseconds).

Model	# vertices	start_stroke	move_stroke	build_layers	draw_layers
Trefoil Knot	144K	168.00	5.73	85.81	0.19
Bottle	190K	330.25	6.03	98.36	0.17
Pegaso	75K	132.05	6.84	67.90	0.12
Children	200K	351.75	11.66	124.83	0.11

USER STUDY

Furthermore, we conducted a user study to quantify the benefits that our system can provide to general users. We compared the proposed *LayerPaint* system against conventional WYSIWYG systems. To make the comparison fair, we consider the following issues when designing the user study: 1) the models to be painted; 2) the painting pattern; and 3) the way to present the requirements to the users. After consulting with several artists, we chose three testing models (see Figure 10) and designed the painting patterns to simulate the common painting tasks.

Another concern is that familiarity with the system may affect the experiments on painting performance. To minimize such a potential discrepancy, we recruited two groups of participants for the user study. The first group painted the required pattern using the conventional system followed by *LayerPaint*, while the other group did exactly the same set of tasks but in reverse order. Before the user study, we briefly informed the participants the controls in the user interface and the tablets. Then, each participant was given five minutes' time to experiment with the painting controls. Note that *LayerPaint* is developed based on an existing WYSIWYG system, so participants can simply check/uncheck a GUI checkbox to enable/disable the layer-aware operations.

There are 15 participants in the first group: eight males and seven females, aged from 22 to 31. Four of them have rich experience in 2D/3D painting and the others are novice painters. We recruited 8 participants in the second group: five males and three females, aged from 21 to 32. Two of them are experienced 2D/3D painters. Note that experienced painters and beginners may have completely different painting styles that could lead to inaccurate measurements in our tests. To reduce such kind of discrepancy, we first marked the required painting region on the testing models (see the colored areas in Figure 10), and the participants were asked to choose a color for the brush and fill the marked region.

There are two stages in the user study. In the first stage (for the first group), the participants were first asked to do a series of painting tasks using a conventional WYSIWYG system. After that, we taught the participants the concept of *LayerPaint* and the related layer-aware operations. Next, they were given another five minutes to experiment with the new features, and then perform the same set of tasks as in the first stage, but with the features in *LayerPaint*. In each painting task, we measured the time taken by each participant to finish the required drawing. The user study for the second group is slightly different. Before the user study, we first taught the participants the concept of 3D painting, *LayerPaint* and the corresponding layer-aware operations. After experimenting with the system for five minutes, the participants were then asked to do the painting tasks using *LayerPaint*. After that, they were given another 5 minutes to get familiar with the painting interface without the layer-aware features, and then perform the same set of tasks with the conventional WYSIWYG interface.

Experiment #1: The first experiment aims at evaluating the performance of painting a long stroke on the *Trefoil Knot* model, see also Figure 10 (left). Note that being able to draw a long stroke can significantly improve the 3D painting efficiency. However, due to occlusions, the participants cannot complete the entire painting with only one stroke when using the conventional system. Instead, they have to carefully rotate and align the model to unveil various hidden regions, and so, multiple strokes are required. Even for experienced users, drawing a line by multiple strokes from different viewpoints can easily lead to unpleasing drawing artifacts like discontinuity, undesired overlap, etc.

Armed with *LayerPaint*, the users can easily finish the required drawing without changing the viewpoint or re-aligning the object. As a result, most participants can finish this task with only a single stroke. The timing statistics show that the longest time taken in this experiment is 93 seconds for conventional system, but only 58 seconds for *LayerPaint*. On average, *LayerPaint* saves 60% of painting time as compared to the conventional system.

Experiment #2: The second experiment is to evaluate the performance of coloring a specified region. The participants were asked to paint on two regions: a front-most region and an inner surface region on Figure 10 (middle). Painting on the front-most layer can be straightforward for both conventional and *LayerPaint* systems, but painting on the inner surface is a real challenge for the conventional system due to blocking visibility. With *LayerPaint*, the users, however, can efficiently paint on the inner surface as quickly as on the front-most layer. As shown in Figure 11, the average time for painting the same region with the conventional system is 2.3 time longer than with *LayerPaint*.

Experiment #3: In contrast to the above two experiments that examine the basic painting operations on synthetic models, this experiment aims at simulating a more non-trivial painting task on a real-world 3D model. The users were re-

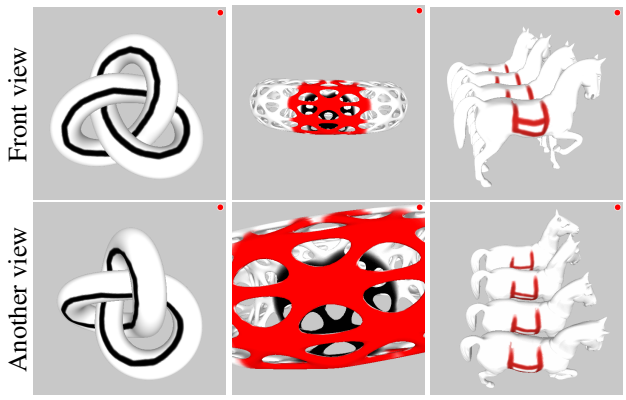

Figure 10. The three 3D models used in the user study: Experiment #1 (left): draw a long stroke on the Trefoil Knot; Experiment #2 (middle): paint on the front-most and inner surface layers on Donuthexa; Experiment #3 (right): draw saddles on the four-horse model.

quired to draw a saddle on each of the horse models. Each saddle needs around four drawing strokes. Note that the horses are aligned one by one, and hence, it is technically very difficult to choose a viewpoint so that the user can view all saddles under the conventional system. Thus, the users can only draw one saddle at a time. But with *LayerPaint*, we can see all saddles at the same viewpoint, see Figure 10 (right), by means of the hidden region popup. As a result, the participants can select a desired layer and quickly paint on it without needing to change the viewpoint or re-align the 3D models. As shown in Figure 11, the average time with *LayerPaint* is only half of that with the conventional system. This implies that *LayerPaint* can significantly shorten the painting time for real painting tasks.

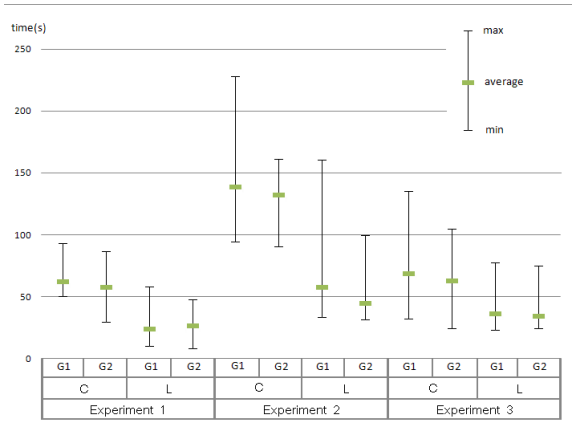

Figure 11. User study results. We measured the time each participant took to complete the task in each experiment: G1 and G2 refer to the first and second groups, respectively, while C and L stand for conventional painting system and *LayerPaint*, respectively. We show the mean time (green bars), maximum time, and minimum time taken for each case. In summary, the timing statistics show that *LayerPaint* outperforms the conventional painting system in all experiments.

Rating. After finishing the above experiments, the participants were required to rate the two test systems on a scale of 1 to 5, where 1 means "not satisfied at all" and 5 means "completely satisfied." Without surprises, *LayerPaint* system receives an average score of 4.9 as compared to 2.0 for the conventional system. The participants also commented that *LayerPaint* is significantly better than the conventional system for painting on invisible layers. However, it also took them more time to learn and understand the layer-aware operations. Considering the advantages we can have with *LayerPaint*, the time and cost for learning are certainly worthy.

Discussions. The experimental results demonstrate that *LayerPaint* is superior over conventional painting interface because it takes advantage of the layer-aware operations to provide more flexibility in 3D painting. For example, painting on inner surfaces (Experiment #2) can be highly efficient with *LayerPaint*, but requires the users to carefully choose the viewpoints in the case of conventional system. The timing statistics also evidence that the users of the conventional system took two to three times longer on average to paint the partially-hidden region (colored in black) than the front-most region (visible region colored in red), though both regions are of similar size. Using *LayerPaint*, the users can easily bring the inner surface to the front and paint on it in exactly the same way as on the frontal regions.

We also would like to point out that *LayerPaint* usually leads to better painting results than the conventional system for invisible/occluded layers. The reason is that both *LayerPaint* and the conventional system project the brush footprint from the screen space onto the tangent space of the surface. A badly chosen viewpoint could lead to a severe distortion during the projection. As demonstrated in the experiments, it is tedious and sometimes highly challenging (or even impossible) to choose a good viewpoint for painting the hidden region with the conventional system. With *LayerPaint*, the users need not worry about the viewpoint and the projection.

Finally, to demonstrate the capability of *LayerPaint* in performing complex tasks, we invited an artist to test our system on Trefoil Knot, see row 1 of Figure 12. It took her 40 minutes to get familiar with the user interface controls and the layer-aware features, and to design and paint on the 3D model. Based on her expertise, we developed additional features such as the flood filling, color picking, etc. And she later also helped us to paint on more 3D models as shown in rows 2 to 5 of Figure 12. The time taken to produce these paintings varies from 20 to 60 minutes.

Limitations.

- *Geometry:* First, *LayerPaint* does not work well for some specific models (which in fact are also hard for conventional systems): 1) models with self-intersections, e.g., Klein bottle; note that *LayerPaint* only considers geometric continuity but not topological connection; 2) noisy models with bumpiness; many tiny layers could result.
- *Graphics Hardware:* Second, *LayerPaint* needs specialized graphics hardware that supports GPU-based fragment processing so as to support its interactive operations.

CONCLUSIONS

In this paper, we proposed *LayerPaint*, a practical, robust, and novel WYSIWYG interface for interactive painting on 3D models. In sharp contrast to the existing WYSIWYG approaches, the paintable area of our approach is not limited to the front-most visible surface on the screen. Thus, the users can efficiently and interactively draw long strokes across different depth layers, and unveil the occluded regions that one would like to see or paint on. Since the painting is depth-sensitive, we can avoid various potential painting artifacts and limitations in conventional WYSIWYG painting interfaces. In addition, *LayerPaint* does not require surface parametrization of the input 3D models and no special haptic device is needed in this approach. Our experimental results, including both the user study and timing statistics, demonstrate the efficacy of the proposed approach, which suggests *LayerPaint* to be a highly efficient and compelling interaction tool for painting real-world 3D models.

ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for the constructive comments given and Sun Qian for her contribution in the 3D painting. This research is supported in part by the following grant projects: MOE AcRF Tier1 Grant (RG 13/08) and NRF2008IDM-IDM-004-006.

REFERENCES

1. M. Agrawala, A. C. Beers, and M. Levoy. 3D painting on scanned surfaces. In *Symp. on Interactive 3D Graphics '95*, pages 145–150, 1995.
2. S.-H. Bae, R. Balakrishnan, and K. Singh. ILoveSketch: as-natural-as-possible sketching system for creating 3D curve models. In *ACM symp. on User interface software and tech.*, pages 151–160, 2008.
3. R. Balakrishnan and G. Kurtenbach. Exploring bimanual camera control and object manipulation in 3D graphics interfaces. In *CHI '99*, pages 56–62, 1999.
4. L. Bavoil and K. Myers. Order independent transparency with dual depth peeling, 2008. White paper, NVidia.
5. D. A. Bowman, E. Kruijff, J. J. LaViola, and I. Poupyrev. *3D User Interfaces: Theory and Practice*. Addison-Wesley Professional, 2004.
6. M. Burns and A. Finkelstein. Adaptive cutaways for comprehensible rendering of polygonal scenes. *ACM Tran. on Graphics (SIGGRAPH Asia)*, 27(5), 2008. Article No. 154.
7. D. DeBry, J. Gibbs, D. D. Petty, and N. Robins. Painting and rendering textures on unparameterized models. In *SIGGRAPH '02*, pages 763–768, 2002.
8. E. Eisemann, S. Paris, and F. Durand. A visibility algorithm for converting 3D meshes into editable 2D vector graphics. *ACM Trans. on Graphics (SIGGRAPH 2009)*, 28(3), 2009. Article No. 83.
9. C. Everitt. Interactive order-independent transparency, 2001. White paper, NVidia.
10. A. Gregory, S. Ehmann, and M. Lin. inTouch: interactive multiresolution modeling and 3D painting with a haptic interface. In *IEEE Virtual Reality*, 2000.
11. C. Grimm and M. A. Kowalski. Painting lighting and viewing effects. In *Intl. Conf. on Comp. Graphics Theory and App.*, pages 204–211, 2007.
12. P. Hanrahan and P. Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. In *SIGGRAPH 90'*, pages 215–223, 1990.
13. A. Khan, B. Komalo, J. Stam, G. Fitzmaurice, and G. Kurtenbach. HoverCam: interactive 3D navigation for proximal object inspection. In *Symp. on Interactive 3D graphics and games*, pages 73–80, 2005.
14. W. Li, M. Agrawala, B. Curless, and D. Salesin. Automated generation of interactive 3D exploded view diagrams. *ACM Tran. on Graphics (SIGGRAPH 2008)*, 27(3), 2008. Article No. 101.
15. W. Li, L. Ritter, M. Agrawala, B. Curless, and D. Salesin. Interactive cutaway illustrations of complex 3D models. *ACM Tran. on Graphics (SIGGRAPH 2007)*, 26(3), 2007. Article No. 31.
16. F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. Single pass depth peeling via CUDA rasterizer. In *SIGGRAPH Asia 2009 Sketch*, 2009.
17. J. McCrae, I. Mordatch, M. Glueck, and A. Khan. Multiscale 3D navigation. In *Symp. on Interactive 3D graphics and games*, pages 7–14, 2009.
18. K. Ryokai, S. Marti, and H. Ishii. I/O brush: drawing with everyday objects as ink. In *CHI '04*, pages 303–310, 2004.
19. K. Ryokai, S. Marti, and H. Ishii. Designing the world as your palette. In *CHI Extended Abs.*, pages 1037–1049, 2005.
20. S. Schkolne, M. Pruett, and P. Schröder. Surface drawing: creating organic 3D shapes with the hand and tangible tools. In *CHI '01*, pages 261–268, 2001.
21. Y. Shon and S. McMains. Evaluation of drawing on 3D surfaces with haptics. *IEEE Comput. Graph. Appl.*, 24(6):40–50, 2004.
22. P. Vandoren, T. Van Laerhoven, L. Claesen, J. Taelman, F. Di Fiore, F. Van Reeth, and E. Flerackers. Dip - it: digital infrared painting on an interactive table. In *CHI Extended Abs.*, pages 2901–2906, 2008.
23. C. Ware and J. Rose. Rotating virtual objects with real handles. *ACM Trans. Comp.-Human Interaction*, 6(2):162–180, 1999.

Appendix: Multi-Layer Segmentation

1) Pixel-level: First of all, we build pixel-level connectivity for each pixel in each depth layer. Basically, this piece of information tells us the depth layer that a pixel connects to for all four 4-connected directions (up, down, left, and right) in the image space. Here we have to determine if a pixel is a boundary pixel on its own layer:

> *A pixel is on boundary if it is not depth-connected to any of the four direct pixel neighbors on the same layer.*

For interactive computation on the GPU, we compute depth-connection between two neighboring pixels by checking whether their depth difference is less than a user-specified threshold, which is set to be 0.001, but could be interactively controlled by the user. In addition, the input model is uniformly pre-scaled to just fit inside a unit cube for consistency. If a given pixel is not a boundary pixel, it can depth-connect to all its four direct neighbors on the same layer; Else, we need to search over all other depth layers for a depth-connection from the pixel for the corresponding 4-connected direction. Since this is a per-pixel operation, we can efficiently perform it by using GPU fragment program, which is performed immediately after the depth peeling. This substep results in an RGBA texture, namely the *connectivity texture*, per depth layer, with each color channel storing the layer ID of the connectable depth layer (if any) for each of the four directions from the pixel, see Figure 3 (left) for an illustration.

2) Segmentation map: After that, we perform a multi-label image segmentation on each connectivity texture to obtain a segmentation map texture, which stores a unique region ID for each segmented region. In detail, a seed-based image segmentation method is used, see Figure 2 for resultant color-mapped segmentation maps.

3) Region-level: Next, we have to build connectivity information among regions across multiple layers, see also Figure 3 (right) for an illustration. In detail, we load all connectivity and segmentation map textures to the main memory, check all boundary pixels (those connected to a layer other than its own), and summarize all these connectivities as the connectivity for each region. As a result, we can obtain a graph data structure with regions as nodes and connections as edges. Note that this substep is now implemented on the CPU, but can also be further put to the GPU by using CUDA.

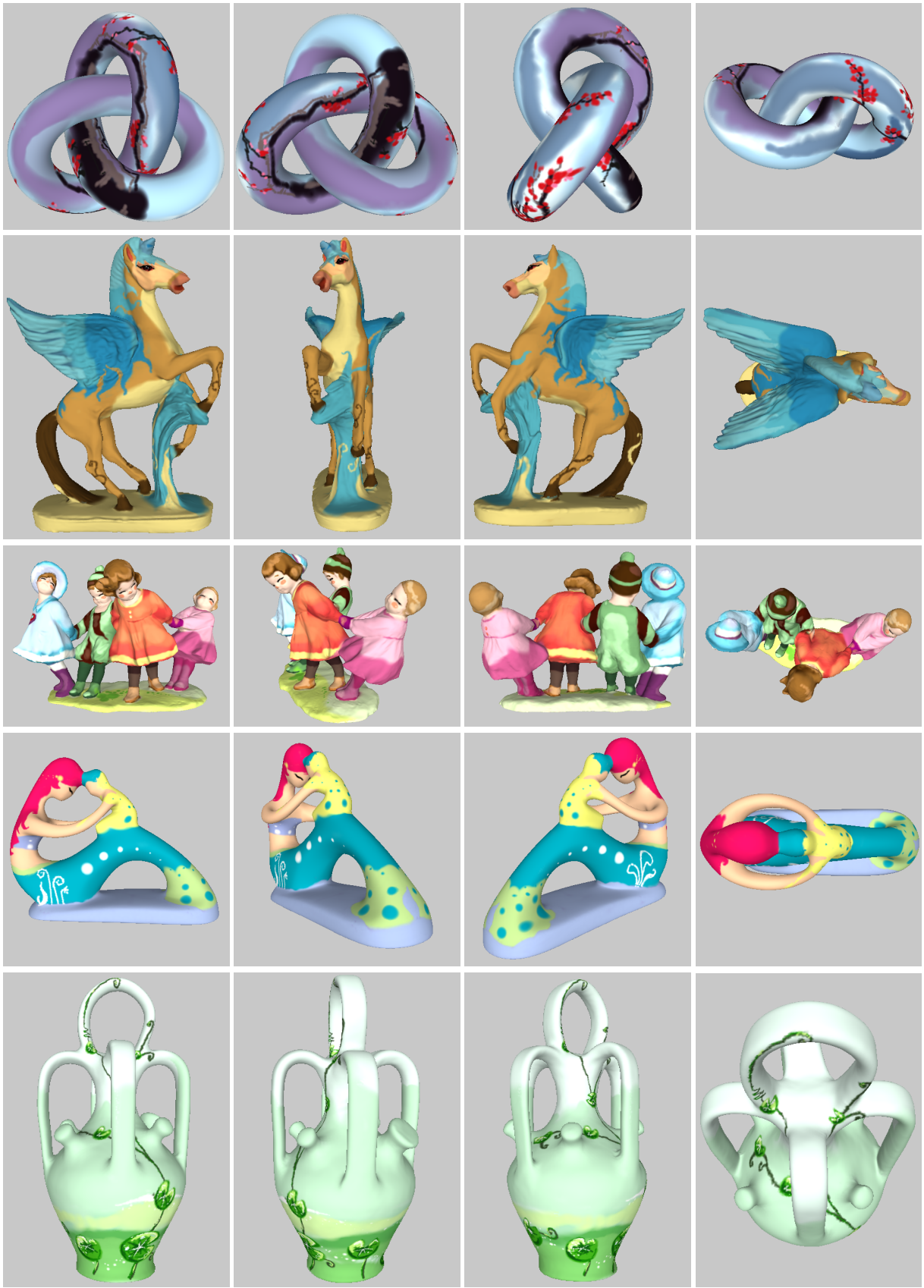

Figure 12. Paintings of 3D models created with *LayerPaint*: *Trefoil Knot*, *Pegaso*, *Children*, *Mother-child*, and *Bottle* (from top to bottom). These paintings were created by an artist at first contact with the tool; painting time varies from 20 to 60 minutes.