Journal of
open research software

nub: A Rendering and Interaction Library for Visual Computing in Processing

SOFTWARE
METAPAPER

JEAN PIERRE CHARALAMBOS

]u[ubiquity press

ABSTRACT

The `nub` library is introduced as an open-source tool for rendering and interaction in the Processing language. Designed for visual computing research, it features a functional and declarative API that simplifies the creation of hierarchical node trees. By decoupling rendering from interaction, the library supports customizable workflows for diverse scenarios, including multi-view scene management, view-frustum culling, gesture-based navigation, and shadow mapping. Seamless integration with Processing ensures compatibility across desktop, Python, and Android platforms. Rich examples and detailed documentation demonstrate its capabilities, making it accessible to researchers, educators, and creative programmers. The framework's extensibility and focus on usability provide a robust foundation for teaching and experimentation in visual computing.

CORRESPONDING AUTHOR:
Jean Pierre Charalambos
Associate Professor, Universidad Nacional de Colombia, Sede Bogotá, CO
jpcharalambosh@unal.edu.co

KEYWORDS:
Processing; 3D graphics; rendering; human computer interaction; rendering and interaction library; scenegraphs; character animation

TO CITE THIS ARTICLE:
Charalambos JP 2025 `nub`: A Rendering and Interaction Library for Visual Computing in Processing. *Journal of Open Research Software,* 13: 4. DOI: https://doi.org/10.5334/jors.477

(1) OVERVIEW

INTRODUCTION

Tree-like affine transformation hierarchies—structures that organize objects in a parent-child relationship with transformations relative to their parent—are essential in rendering, interaction, and computer vision, all core disciplines of visual computing. These hierarchies enable hierarchical space partitioning, a process that recursively divides space into non-overlapping subspaces, forming the foundation for advanced real-time rendering techniques such as view-frustum and occlusion culling [21, 3, 34], ray-tracing [31, 20, 8], multiresolution meshes [9, 18], and collision detection [32, 15].

Additionally, they support the creation of compound objects, enabling key interaction tasks like navigation, picking, and spatial manipulation [12]. They also form the basis of articulated structures, supporting applications such as inverse kinematics [5], and (non-)human skeletal animation and motion retargeting [4, 19, 2]. These tasks are central to post-WIMP (Window, Icon, Menu, Pointer) user interfaces, which transcend traditional graphical paradigms by incorporating 3D interaction, gestures, and immersive environments [17, 27].

This paper introduces the `nub` library, a lightweight, extensible framework for setting up node trees—acyclic hierarchies rooted at a single `null` root—designed for rendering and interaction experiments. The library integrates seamlessly with Processing, a flexible Java-based language for creative programming in visual design, widely used for teaching, prototyping, and computational art [26, 29, 25, 7]. Processing provides a simple syntax, extensible architecture, and vibrant community, making it an ideal foundation for tools like `nub`, which builds on its core principles to support advanced rendering and interaction workflows. Its key features include:

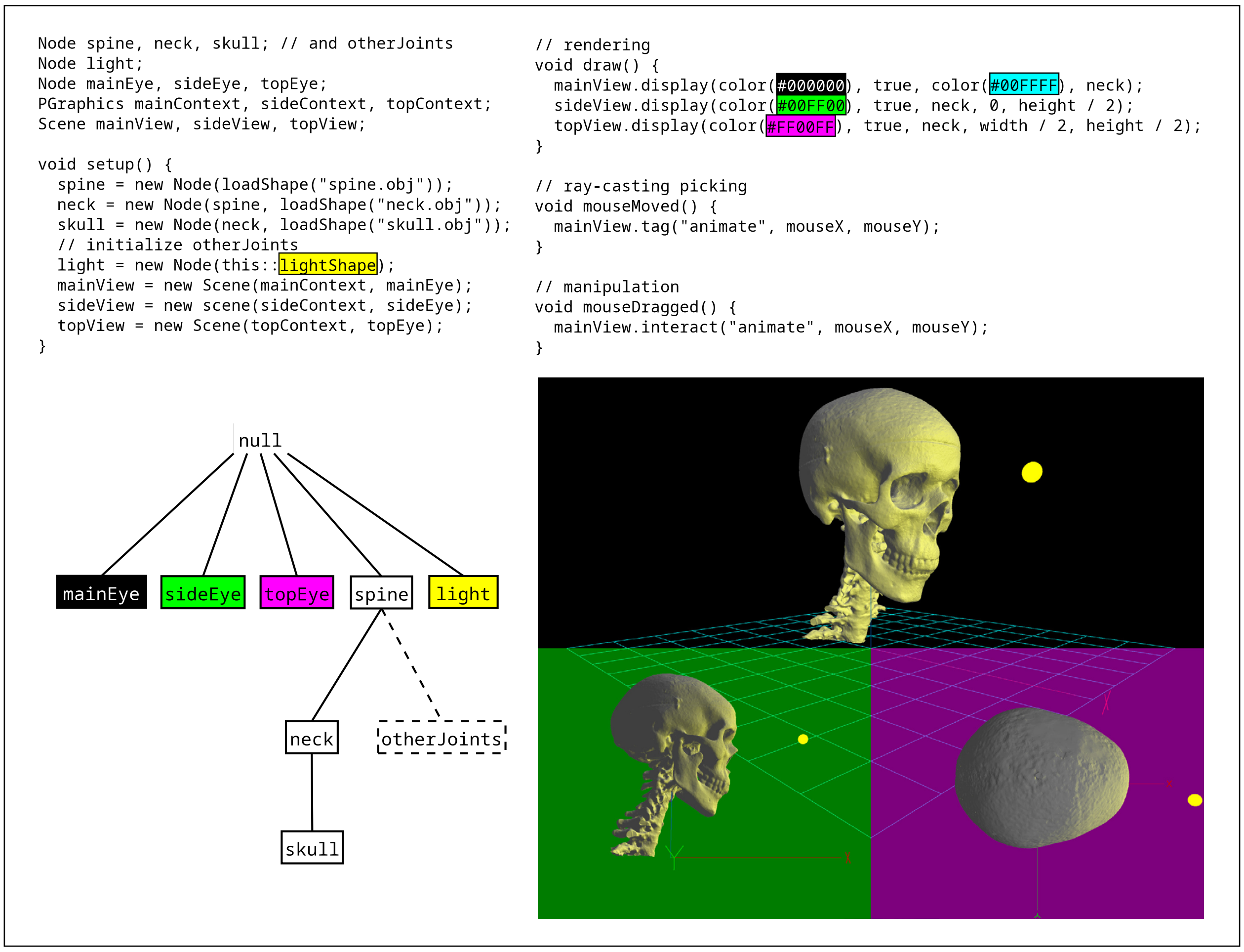
- **Simplicity and Portability:** The library has a simple design with no external dependencies, exposing functionality through a high-level, declarative API centered on the *Node* and *Scene* classes. This design promotes API learnability and portability [30].
- **Customizable Rendering Scenarios:** Users can define custom spatial subdivision algorithms during hierarchy creation—such as Bounding Volume Hierarchies (BVHs), Binary Space Partitioning (BSP) trees, or octrees (see the *Octree* code example below)—and customize culling criteria during traversal, supporting a wide range of advanced real-time rendering techniques [1].
- **Decoupled Interactivity:** The library explicitly exposes interaction through concise, high-level patterns by directly handling interactivity through input devices (e.g., Processing's *mouseMoved* or *mouseDragged* functions), enabling easy customization with defaults for motion actions. By decoupling rendering from interaction, it allows for easily configurable input-handling setups without reliance on agents and hidden profiles, unlike frameworks such as `Proscene` [6].
- **Seamless Processing Integration:** `nub` integrates smoothly with Processing, supporting desktop, Python, and Android 2D/3D modes [26].

By combining a simple and extensible design, seamless integration, decoupling of interaction from rendering, and customizability of rendering scenarios, the `nub` library provides a robust foundation for visual computing experiments.

IMPLEMENTATION AND ARCHITECTURE

A `nub` visual application consists of a static node tree and one or more scenes, each encapsulating an *eye* node which defines view transformations during rendering, and a rendering context used to render node (sub)trees (see Figure 1). All of the main framework functionality is built on just two core classes:

1. ***Node:*** This class represents a 2D or 3D subspace and caches a composed affine transformation comprising a translation followed by a rotation, and a uniform positive scaling. It can have *inertia*, *filters* to constrain motion, and *keyframes* to model it [35]. Nodes may be rendered according to a visual *hint*, with details provided in the repository's readme manual.
   Node instances are organized into a static tree whose root is the `null` reference, as depicted in Figure 1 (left). Nodes support precise picking through their screen projection hints. Methods for node localization and spatial transformations are also described in the repository's readme manual.
2. ***Scene:*** This class acts as a wrapper for a Processing *PGraphics*, which serves as the rendering context for the node tree. Each scene encapsulates an *eye* node to define view transformations and caches its own *view*, *projection*, *projectionView*, and *projectionViewInverse* matrices. It also provides routines for visibility checks, including *point*, *box*, and *ball* visibility (see the *ViewFrustumCulling* code example below). Additionally, it gathers input data from Processing's input methods or other sources, such as multi-touch gestures and game controllers, and applies interaction patterns to implement custom interactive actions for nodes.
   Scene methods for transforming between world, screen, and normalized-device-coordinates (NDC) spaces are detailed in the repository's readme manual.



```
Node spine, neck, skull; // and otherJoints
Node light;
Node mainEye, sideEye, topEye;
PGraphics mainContext, sideContext, topContext;
Scene mainView, sideView, topView;

void setup() {
  spine = new Node(loadShape("spine.obj"));
  neck = new Node(spine, loadShape("neck.obj"));
  skull = new Node(neck, loadShape("skull.obj"));
  // initialize otherJoints
  light = new Node(this::lightShape);
  mainView = new Scene(mainContext, mainEye);
  sideView = new scene(sideContext, sideEye);
  topView = new Scene(topContext, topEye);
}
```

```
// rendering
void draw() {
  mainView.display(color(#000000), true, color(#00FFFF), neck);
  sideView.display(color(#00FF00), true, neck, 0, height / 2);
  topView.display(color(#FF00FF), true, neck, width / 2, height / 2);
}

// ray-casting picking
void mouseMoved() {
  mainView.tag("animate", mouseX, mouseY);
}

// manipulation
void mouseDragged() {
  mainView.interact("animate", mouseX, mouseY);
}
```

Figure 1 nub main framework elements: Left: a static *Node* tree implementing a multi-view application for a human skeleton and defining shape visual *hints* which specify what to draw for each node (*spine*, *neck*, *skull*, and *light*). The tree supports three *scenes* (*mainView*, *sideView*, and *topView*), each encapsulating an *eye Node* (*mainEye*, *sideEye*, and *topEye*, respectively) and a *PGraphics* rendering context (*mainContext*, *sideContext*, and *topContext*, respectively); Right: rendering of the *neck* subtree from each scene's eye viewpoint, demonstrating ray-casting picking and manipulation. The 3D model of the human skeleton was adapted from [14].

RENDERING

The scene's *render([subtree])* and *display([background], [axes], [grid], [subtree], [worldCallback], [pixelX], [pixelY])* methods support diverse rendering scenarios, as illustrated in Figure 1 (right) and Figure 2.

The *render([subtree])* method traverses the node *subtree* hierarchy, rendering each node's visual hint onto the scene rendering context. It also supports defining custom node behaviors, such as periodic tasks, which are executed during the traversal. These behaviors can be specified using the *setBehavior(Node node, Consumer<Node> behavior)* method.

The *display([background], [axes], [grid], [subtree], [worldCallback], [pixelX], [pixelY])* method follows a three-step process: first, it fills the *background* and displays world *axes* and *grid*; second, it invokes the *render([subtree])* method and the user-provided *worldCallback* function; finally, it displays the rendered scene context at the screen position *(pixelX, pixelY)* which specifies its top-left corner.

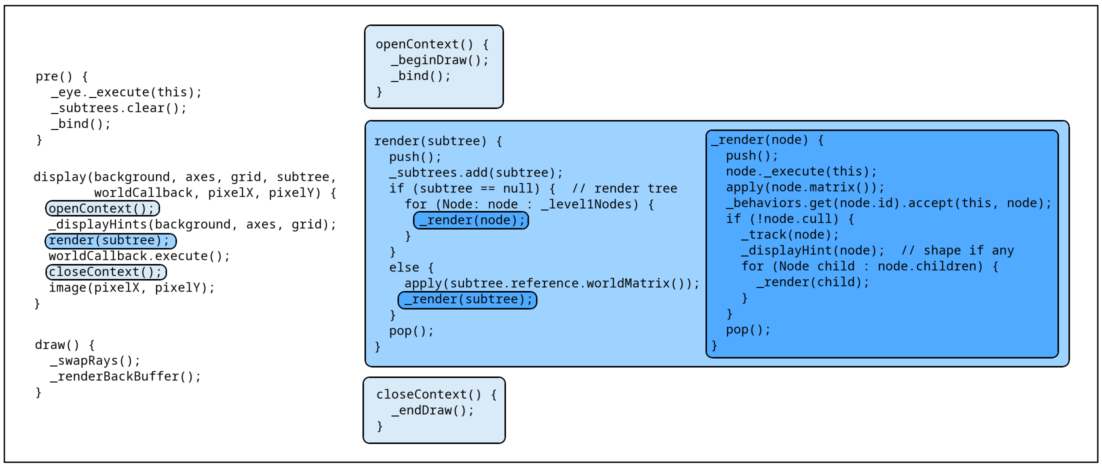
The above methods handle rendering scenarios ranging from simple to complex, including rendering the same node tree onto different rendering contexts from various viewpoints (e.g., for *bokeh* effects [22, 28, 33]) or supporting 2D/3D interactive minimaps and shadow mapping (see the *MiniMap* code example below).

INTERACTION

All use-case interaction scenarios are handled by the scene's *interact(node, gesture)*, *interact(tag, gesture)*, and *interact(gesture)* patterns.

The *interact(node, gesture)* pattern sends gesture data to the specified node, or to the eye when the node parameter is *null*. This triggers a user-provided functor, defined by the node's *setInteraction(Consumer<Object[]> functor)* method, which implements the action to be performed by the node (see the *CustomNodeInteraction* code example below).

The *interact(tag, gesture)* pattern enables node picking and manipulation by resolving the node parameter using the *node(tag)* method. This is equivalent to calling *interact(node(tag), gesture)*, as illustrated in Figure 1 (right). Nodes can be tagged using *tag([tag], node)* or



```
pre() {
  _eye._execute(this);
  _subtrees.clear();
  _bind();
}

display(background, axes, grid, subtree,
        worldCallback, pixelX, pixelY) {
  openContext();
  _displayHints(background, axes, grid);
  render(subtree);
  worldCallback.execute();
  closeContext();
  image(pixelX, pixelY);
}

draw() {
  _swapRays();
  _renderBackBuffer();
}
```

```
openContext() {
  _beginDraw();
  _bind();
}
```

```
render(subtree) {
  push();
  _subtrees.add(subtree);
  if (subtree == null) {  // render tree
    for (Node: node : _level1Nodes) {
      _render(node);
    }
  }
  else {
    apply(subtree.reference.worldMatrix());
    _render(subtree);
  }
  pop();
}
```

```
_render(node) {
  push();
  node._execute(this);
  apply(node.matrix());
  _behaviors.get(node.id).accept(this, node);
  if (!node.cull) {
    _track(node);
    _displayHint(node);  // shape if any
    for (Node child : node.children) {
      _render(child);
    }
  }
  pop();
}
```

```
closeContext() {
  _endDraw();
}
```

Figure 2 *render([subtree])* and *display([background], [axes], [grid], [subtree], [worldCallback], [pixelX], [pixelY])* scene methods. The *render* method, which may be called multiple times per frame (e.g., with specific ordering to render node subtrees), recursively renders the node *subtree* onto the scene context and collects the subtrees to be rendered onto the backbuffer (*_renderBackBuffer*) for resolving picking in the next frame (*_track*). The *display* method renders the *axes*, *grid* hints, and node tree (*render*) on the screen. Setting up the rendering context's view and projection matrices according to the eye (*_bind*) is automatically handled by the Processing *pre* registered function (for onscreen scenes) or by the *display* method (for offscreen scenes). This setup can also be explicitly invoked multiple times to *render* offscreen scenes using *openContext* and *closeContext*, which call Processing's *beginDraw* and *endDraw* on the scene context, respectively. Displaying an offscreen-rendered scene requires invoking the scene's high-level *image(pixelX, pixelY)* (top-left corner), which is automatically handled by *display*. Additionally, rays used for picking, gathered during the current and previous frames (e.g., through mouse events), are swapped in Processing *draw*, leveraging temporal coherence similarly to double buffering, but at a much lower computational cost.

by performing ray-casting against the node's visual hint with *tag([tag], [pixelX], [pixelY])*.

The *interact(gesture)* pattern sends gesture data directly to the eye unless a node with the *null* tag exists, in which case the node becomes the interaction target.

Default motion actions for both the eye and nodes, derived from screen-space gesture data, are included in `nub` and detailed in the code repository's readme manual.

These interaction patterns support a wide spectrum of interactive scenarios, ranging from simple to highly complex. They are compatible with various input devices, including those with multiple degrees of freedom (see the *CustomNodeInteraction* code example below).

QUALITY CONTROL

The `nub` library employs a comprehensive testing framework, publicly available at https://github.com/VisualComputing/nub/tree/master/testing, which integrates unit and regression tests. For every new feature implementation, dedicated examples are created to rigorously test the introduced functionality. This approach not only ensures the robustness of new features but also preserves the overall integrity of the library.

After feature-specific testing, all existing examples are subjected to regression tests to identify and address any unintended side effects or alterations to the library's existing functionality. Once these tests are successfully completed, selected examples are integrated into the Processing release of the library, initiating a new cycle of development and testing.

Users can report issues on GitHub for additional support with the library, and contributions from the community are actively encouraged to enhance its development.

(2) AVAILABILITY

OPERATING SYSTEM

`nub` is platform-agnostic, enabling it to run on any operating system (GNU/Linux, macOS, Android, Windows) that supports Processing.

PROGRAMMING LANGUAGE

`nub` v1.1.1 runs on Processing 4.2 and later.

DEPENDENCIES

`nub` has no dependencies other than Processing 4.2 and later. In addition, `nub` can easily be installed using the Processing IDE import library utility.

SOFTWARE LOCATION

Code repository *GitHub*

Name: `nub`

Persistent identifier: https://doi.org/10.5281/zenodo.8033963

Licence: GPL-v3

Version published: 1.1.1

Date published: 13/6/23

The first version of `nub`, previously known as `frames` during its proof-of-concept stage, was published on GitHub on 25/09/2019.

LANGUAGE

Processing

(3) REUSE POTENTIAL

nub is a library developed for Processing, a widely used open-source programming language, and designed to be accessible to a diverse audience within the visual computing community. Its *Node* class is decoupled from Processing, possibly enabling integration with other frameworks by implementing a *Scene* interface specific to the target framework.

The nub API, fully documented at https://visualcomputing.github.io/nub-javadocs/, provides extensive customization capabilities for core library features. The functional and declarative API leverages modern Java features, eliminating the need for class inheritance and enhancing simplicity. The library has been applied in various research fields, including inverse kinematics [5].

Feedback can be submitted through the GitHub issue tracker or via email to support the library's development and usability.

CODE EXAMPLES

The current nub release includes several examples highlighting different aspects of the library. A selection of these examples, illustrating the library's capabilities, is detailed below (see Figure 3):

(a) A forward kinematics-based scene featuring Pixar's iconic *Luxo* [16];
(b) A scene showcasing a cube and an eye with keyframe hints;
(c) A depth-map rendered onto an offscreen context from an arbitrary box viewpoint;
(d) A shadow mapping scene [11, 24, 23], requiring a depth-map of the node hierarchy rendered from the light's viewpoint (building on the previous example);
(e) A scene with multiple toric solenoids parsing gesture data to modify their topology; and,

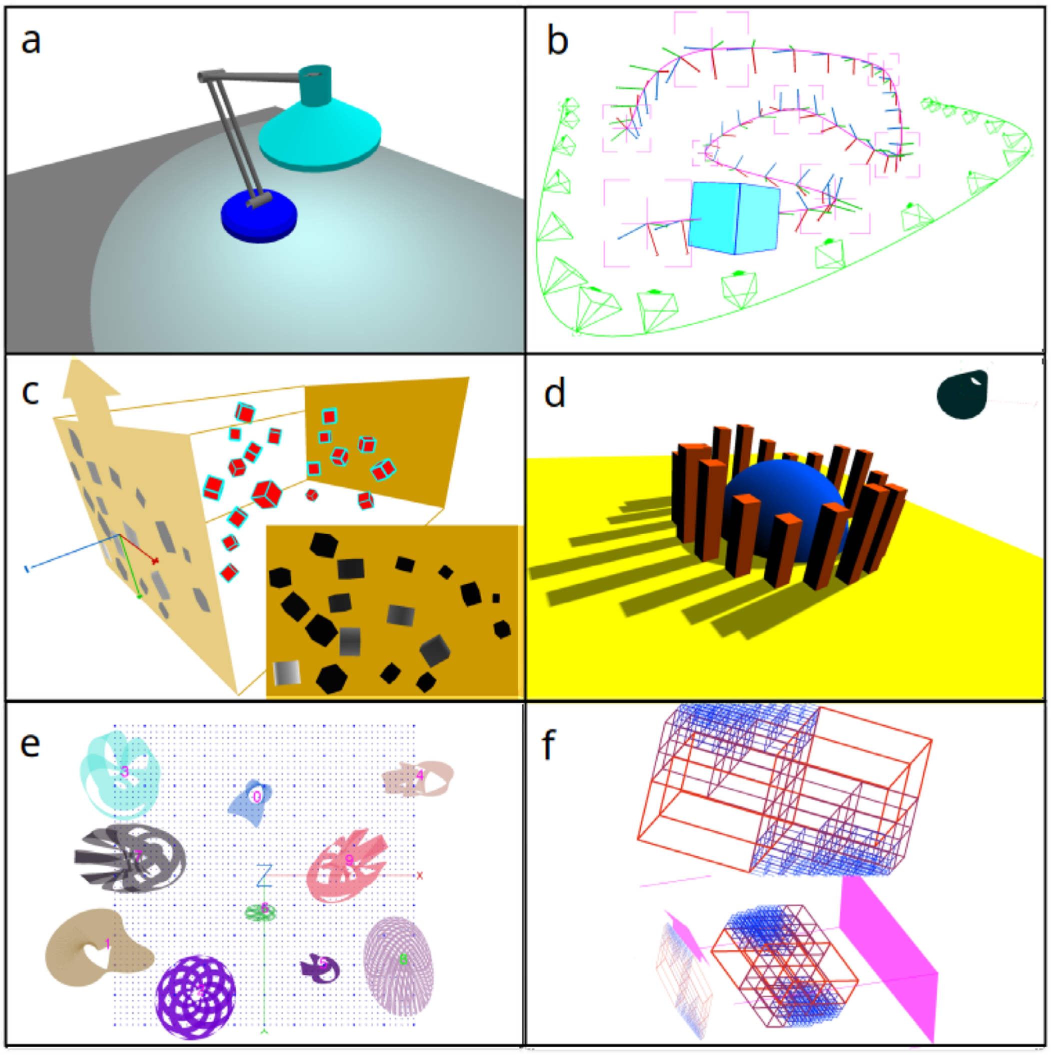


Figure 3 nub selected examples: **a)** Luxo; **b)** Keyframes; **c)** Depth map; **d)** Shadow mapping; **e)** Custom node interaction; and, **f)** View frustum culling.

(f) An octree scene culled against the viewing volume [13, 10].

ACKNOWLEDGEMENTS

Gratitude is extended to the anonymous reviewers for testing the software, running the examples, and reviewing the API documentation, as well as for their valuable suggestions. Thanks are also due to Sebastian Chaparro for his thorough insights and for releasing several `nub`-based experiments as free software, and to all Processing contributors.

FUNDING INFORMATION

No funding was received for this work.

COMPETING INTERESTS

The author has no competing interests to declare.

AUTHOR AFFILIATIONS

Jean Pierre Charalambos orcid.org/0000-0002-0945-3829
Associate Professor, Universidad Nacional de Colombia, Sede Bogotá, CO

REFERENCES

1. **Akenine-Möller T, Haines E, Hoffman N, Pesce A, Iwanicki M, Hillaire S.** *Real-Time Rendering* 4th Edition. A K Peters/CRC Press, Boca Raton, FL, USA; 2018. DOI: https://doi.org/10.1201/b22086
2. **Andreou N, Aristidou A, Chrysanthou Y.** Pose representations for deep skeletal animation. *Computer Graphics Forum*. 2022; 41(8): 155–167. DOI: https://doi.org/10.1111/cgf.14632
3. **Beyer J, Hadwiger M, Pfister H.** State-of-the-art in gpu-based large-scale volume visualization. *Computer Graphics Forum*. 2015; 34(8): 13–37. DOI: https://doi.org/10.1111/cgf.12605
4. **Biswas S, Yin K, Shugrina M, Fidler S, Khamis S.** Hierarchical neural implicit pose network for animation and motion retargeting. *ArXiv*, abs/2112.00958; 2021.
5. **Chaparro S.** Master Thesis. Método de cinemática inversa en tiempo real basado en FABRIK para estructuras altamente restrictas; 2021. https://repositorio.unal.edu.co/handle/unal/79872.
6. **Charalambos JP.** Proscene: A feature-rich framework for interactive environments. *SoftwareX*. 2017; 6: 48–53. DOI: https://doi.org/10.1016/j.softx.2017.01.002
7. **Colubri A, Fry B.** Introducing processing 2.0. In *ACM SIGGRAPH 2012 Talks, SIGGRAPH 2012*, New York, NY, USA. Association for Computing Machinery; 2012. DOI: https://doi.org/10.1145/2343045.2343061
8. **Deng Y, Ni Y, Li Z, Mu S, Zhang W.** Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. *ACM Computing Surveys*. 08, 2017; 50: 1–41. DOI: https://doi.org/10.1145/3104067
9. **Du Z, Chiang Y-J.** Out-of-core simplification and crack-free lod volume rendering for irregular grids. *Computer Graphics Forum*. 2010; 29(3): 873–882. DOI: https://doi.org/10.1111/j.1467-8659.2009.01705.x
10. **Friston S, Dobos J, Wong C, Fan C, Montero S, Steed A.** Rectangular selection of components in large 3d models on the web. In *The 24th International Conference on 3D Web Technology, Web3D '19*. New York, NY, USA; 2019. pages 1–9. ACM. DOI: https://doi.org/10.1145/3329714.3338125
11. **Gumbau J, Sbert M, Szirmay-Kalos L, Chover M, González C.** Shadow map filtering with gaussian shadow maps. In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry, VRCAI '11*. New York, NY, USA; 2011. pages 75–82. ACM. DOI: https://doi.org/10.1145/2087756.2087766
12. **Jankowski J, Hachet M.** A Survey of Interaction Techniques for Interactive 3D Environments. In *Eurographics 2013 – STAR*. Girona, Spain; May 2013.
13. **Jurado JM, Graciano A, Ortega L, Feito FR.** Web-based gis application for real-time interaction of underground infrastructure through virtual reality. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL '17*. New York, NY, USA; 2017. pages 97: 1–97: 4. ACM. DOI: https://doi.org/10.1145/3139958.3140004
14. **Kaukonen L.** Human skull and neck; 2019. https://sketchfab.com/3d-models/human-skull-and-neck-a47ef69ff3a4402783cff0f841bc5e0a. Available under the CC Attribution license.
15. **Kim YJ, Lin MC, Manocha D.** *Collision Detection*. Springer Netherlands, Dordrecht; 2018. pages 1–24. DOI: https://doi.org/10.1007/978-94-007-7194-9_26-1
16. John. Lasseter. Luxo jr. film; 1986.
17. **LaViola Jr JJ, Kruijff E, McMahan RP, Bowman D, Poupyrev IP.** *3D user interfaces: theory and practice*. Addison-Wesley Professional; 2017.
18. **Li J, Wu H, Yang C, Wong DW, Xie J.** Visualizing dynamic geosciences phenomena using an octree-based view-dependent lod strategy within virtual globes. *Computers & Geosciences*. 2011; 37(9): 1295–1302. DOI: https://doi.org/10.1016/j.cageo.2011.04.003
19. **Li Z, Zheng Z, Wang L, Liu Y.** Animatable gaussians: Learning pose-dependent gaussian maps for high-fidelity human avatar modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*; 2024.

20. **Maria M, Horna S, Aveneau L.** Constrained convex space partition for ray tracing in architectural environments. *Computer Graphics Forum*. 2017; 36(1): 288–300. DOI: https://doi.org/10.1111/cgf.12801
21. **Mattausch O, Bittner J, Jaspe A, Gobbetti E, Wimmer M, Pajarola R.** Chc+rt: Coherent hierarchical culling for ray tracing. *Computer Graphics Forum*. 2015; 34(2): 537–548. DOI: https://doi.org/10.1111/cgf.12582
22. **McIntosh L, Riecke BE, DiPaola S.** Efficiently simulating the bokeh of polygonal apertures in a post-process depth of field shader. *Comput. Graph. Forum*. September 2012; 31(6): 1810–1822. DOI: https://doi.org/10.1111/j.1467-8659.2012.02097.x
23. **Peters C.** Non-linearly quantized moment shadow maps. In *Proceedings of High Performance Graphics, HPG '17*. New York, NY, USA; 2017. pages 15:1–15:11 ACM. DOI: https://doi.org/10.1145/3105762.3105775
24. **Peters C, Klein R.** Moment shadow mapping. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games, i3D '15*. New York, NY, USA; 2015. pages 7–14. ACM. DOI: https://doi.org/10.1145/2699276.2699277
25. **Reas C, Fry B.** *Make: Getting Started with Processing*. Maker Media, Inc.; 2010.
26. **Reas C, Fry B.** *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press, Cambridge, Massachusetts; 2014.
27. **Riecke BE, LaViola JJ, Kruijff E.** 3d user interfaces for virtual reality and games: 3d selection, manipulation, and spatial navigation. In *ACM SIGGRAPH 2018 Courses, SIGGRAPH 2018*, New York, NY, USA. Association for Computing Machinery; 2018. DOI: https://doi.org/10.1145/3214834.3214869
28. **Selgrad K, Reintges C, Penk D, Wagner P, Stamminger M.** Real-time depth of field using multi-layer filtering. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games, i3D '15*. New York, NY, USA; 2015. pages 121–127. ACM. DOI: https://doi.org/10.1145/2699276.2699288
29. **Shiffman D.** *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction*. Morgan Kaufmann, 2nd edition; 2015.
30. **Stylos J, Myers BA.** The implications of method placement on api learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16*. New York, NY, USA; 2008. pages 105–112. ACM. DOI: https://doi.org/10.1145/1453101.1453117
31. **Vinkler M, Havran V, Bittner.** Performance comparison of bounding volume hierarchies and kd-trees for gpu ray tracing. *Computer Graphics Forum*. 2016; 35(8): 68–79. DOI: https://doi.org/10.1111/cgf.12776
32. **Wang X, Tang M, Manocha D, Tong R.** Efficient bvh-based collision detection scheme with ordering and restructuring. *Computer Graphics Forum*. 2018; 37(2): 227–237. DOI: https://doi.org/10.1111/cgf.13356
33. **Weier M, Roth T, Hinkenjann A, Slusallek P.** Foveated depth-of-field filtering in head-mounted displays. *ACM Trans. Appl. Percept*. September 2018; 15(4): 26: 1–26: 14. DOI: https://doi.org/10.1145/3238301
34. **Xue J, Zhao G, Xiao W.** An efficient gpu out-of-core framework for interactive rendering of large-scale cad models. *Computer Animation and Virtual Worlds*. 2016; 27(3–4): 231–240. DOI: https://doi.org/10.1002/cav.1704
35. **Yuksel C, Schaefer S, Keyser J.** Parameterization and applications of catmull-rom curves. *Comput. Aided Des.* July 2011; 43(7): 747–755. DOI: https://doi.org/10.1016/j.cad.2010.08.008

TO CITE THIS ARTICLE:
Charalambos JP 2025 nub: A Rendering and Interaction Library for Visual Computing in Processing. *Journal of Open Research Software,* 13: 4. DOI: https://doi.org/10.5334/jors.477

Submitted: 15 June 2023 **Accepted:** 20 March 2025 **Published:** 27 March 2025