PyBatchRender：一款可实现高达每秒一百万帧的 Python 批量三维渲染库

Evgenii Rudakov[1], **Jonathan Shock**[2, 3, 4, 5], **Benjamin Ultan Cowley**[1, 6]赫尔辛基大学教育科学学院[2]开普敦大学数学与应用数学系[3]开普敦大学神经科学研究所[4]INRS，加拿大蒙特利尔[5]NiTheCS，南非斯泰伦博斯[6]赫尔辛基大学数字人文系认知科学，evgenii.rudakov@helsinki.fi、jonathan.shock@uct.ac.za、ben.cowley@helsinki.fi

arXiv:2601.01288v1 [cs.GR] 3 Jan 2026

摘要

基于像素的强化学习往往受限于三维渲染环境的性能和复杂性。研究人员不得不在高速、底层的引擎与速度较慢但更易用的 Python 框架之间权衡。为此，我们推出了 PyBatchRender，这是一款高吞吐量的批量三维渲染 Python 库，在简单场景中可实现超过每秒 100 万帧的渲染速度。它基于 Panda3D 游戏引擎构建，利用其成熟的生态系统，并通过优化的批量渲染提升性能，最高可实现 1000× 倍加速。PyBatchRender 专为基于像素的强化学习打造，不依赖特定物理引擎；与专用库相比，它提供了更大的灵活性，设置也比典型的游戏引擎封装更简单，速度可媲美 Madrona 等最先进的 C++ 引擎。用户只需编写几十行 Python 代码，就能创建自定义场景，从而快速构建可扩展的人工智能训练原型。该库采用开源方式发布，易于集成，旨在让研究人员和开发者都能使用高性能三维仿真。该库可在 https://github.com/dolphin-in-a-coma/PyBatchRender 获取。

引言

基于像素的强化学习（RL）是一个具有重要现实应用价值的关键领域，涵盖具身人工智能和电子游戏等方面的研究。然而，与基于向量的环境相比，直接从视觉观测中进行训练面临着诸多挑战。基于像素的方法样本效率显著更低，需要比直接使用向量观测多出几个数量级的训练数据（Yarats等，2019）。渲染成本进一步加剧了这一挑战。每个像素观测所需的计算资源都显著多于基于状态的观测。在复杂的三维渲染场景中，这种性能差距尤其严重：将画面渲染为像素的速度可能比物理步进慢100倍（Zakka等，2025）。这些挑战促使人们寻求优化渲染的方法。

直到最近，专用数据导向渲染的问题仍未得到解决，使其成为基于像素的强化学习的主要瓶颈。人们已尝试针对数据密集型流水线优化渲染，但由于复杂性较高，这些方案并未得到广泛采用（Shacklett 等，2021；Berges 等，2023）。

一些模拟器扩展了物理引擎，加入渲染能力，但视觉吞吐量提升有限。Isaac Lab（基于 NVIDIA 的 Isaac Sim）（Mittal 等，2023）和 ManiSkill3（基于开源 SAPIEN 框架）（Mu 等，2021；Tao 等，2025）代表了将物理仿真与渲染相结合的 GPU 并行化方案。这两个框架在简单环境中均可达到每秒 5 万帧的速度。然而，它们仍与各自的物理引擎紧密耦合，且主要针对机器人操作进行优化，而非通用渲染。

Madrona代表了高吞吐渲染领域的一项重大进展（Shacklett等，2023；Rosenzweig等，2024）。最近，Madrona渲染被集成到MuJoCo Playground中（Zakka等，2025），在3D CartPole等基准测试中实现了每秒403k步。尽管Madrona能够提供令人印象深刻的性能，但其复杂度相对较高。Madrona要求使用C++编写环境规范，这在实践中的强化学习领域构成了障碍，因为相关工作流程往往基于Python。值得注意的是，Madrona及其前身也选择了从头开发面向强化学习的环境，而不是依赖成熟引擎所带来的便利。

传统游戏引擎也经过改造，用于强化学习。Unity（搭配 ML-Agents）（Unity Technologies 2025年）和 Godot（搭配 godot-rl）（Beeching 等，2021年）等平台提供了丰富的渲染和物理功能。与专用渲染器相比，这些解决方案更易于使用，但用户需要在游戏引擎与训练脚本之间建立通信协议。这会造成通信瓶颈，通常使简单环境的仿真吞吐量限制在每秒 2k–4k 帧（FPS）（Beeching 等，2021 年）。

Panda3D 是一款面向 Python 的游戏引擎（Goslin 和 Mine，2004），凭借其易用性，逐渐受到 Python 开发者的青睐。它的灵活性使其成为复杂驾驶模拟器（如 MetaDrive（Li 等人（2023））以及多种科学模拟器的基础（Kiefl，2024；Goslin 和 Mine，2004）。其优势包括可通过着色器进行底层控制，且没有僵化的结构。然而，Panda3D 最初并非为大规模批量渲染而设计，因此在典型的强化学习训练循环中，吞吐量并不高（通常为每秒 1k—

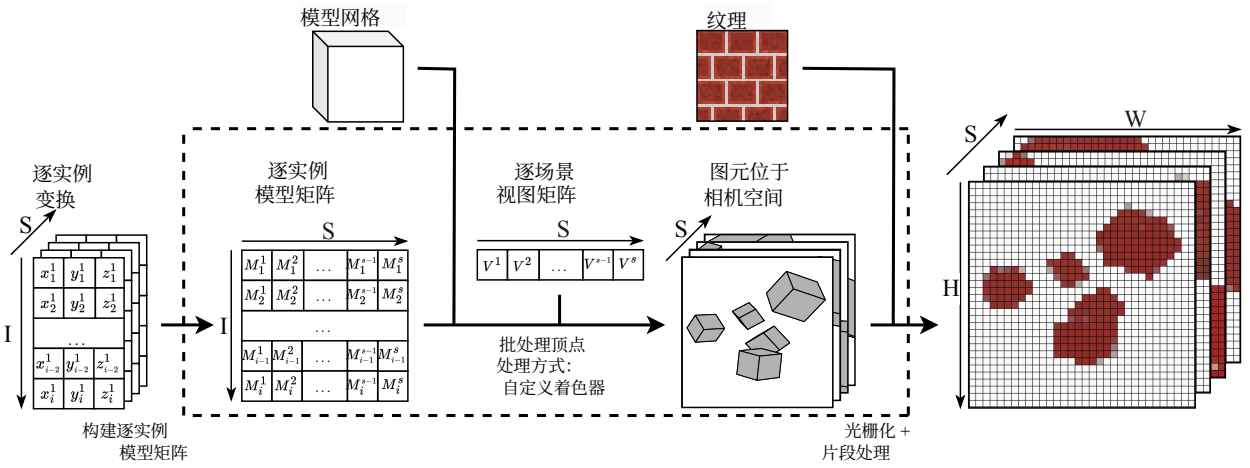

图1：在多个场景中对单个几何体进行批量渲染的管线。用户为跨越 S 个场景的共享网格中的 I 个实例指定逐实例平移（虚线框将用户控制与内部操作区分开）。系统构建逐实例模型矩阵 M_i^s，将其与逐场景视图矩阵 V^s 组合，并使用自定义批量顶点着色器，在单次渲染通道中处理所有场景中的所有实例。光栅化和片段处理生成一张平铺输出纹理，随后将其划分为 S 个分辨率为 $H \times W$ 的独立帧，这些帧保留在图形处理器上，以供后续访问。这种方法将 $S \times I$ 个独立渲染通道整合为一次并行化的图形处理器操作，消除了渲染目标切换开销，并最大限度地提高了硬件利用率。未显示：逐实例缩放、旋转和颜色，以及通过位置/方向等方式进行的相机控制；系统均支持这些功能，但为清晰起见，此处未予展示。

每秒 2k 帧（Li 等人（2023））。

这一领域呈现出明显的权衡：研究人员必须在 C++ 专用引擎的高性能但高复杂度，与传统游戏引擎封装的易用性但低性能之间做出选择。

为弥合易用性与高性能之间的差距，我们推出了 PyBatchRender，这是一款面向数据的 3D 渲染库。该库旨在实现简单安装、使用纯 Python 模块化地指定环境，并提供高吞吐量性能。在简单几何体环境中，无论是在数据中心 GPU 还是消费级 GPU 上，它每秒都能渲染超过 100 万帧。

方法

PyBatchRender 的方法不同于以往面向数据生成和强化学习的渲染器。我们没有从头开发渲染引擎，而是在成熟的 Panda3D 游戏引擎之上构建了一个专注于性能的接口。这一架构决策使我们能够专注于批量渲染中的关键瓶颈，包括批量场景控制和像素数据采集。同时，我们将传统渲染操作交由 Panda3D 已优化的实现处理。除此之外，这种方法还便于将现有的 Panda3D 功能（包括资源加载和着色器管线）直接应用于高吞吐量渲染场景。

Panda3D 为三维渲染提供了全面的基础支持，可处理模型加载、纹理管理、光栅化和场景图遍历。除了这些高级功能外，它还通过自定义 GLSL 着色器提供底层控制，使开发者能够精细操控各个实例的绘制方式。然而，Panda3D 的应用程序接口是为传统游戏开发而设计的，在这类开发中，物体和摄像机通过 Python 函数调用逐个进行操作。管理数千个并行环境时，这种设计会产生大量开销，因为每个物体都需要单独调用应用程序接口，无法高效地进行向量化。

我们通过一系列有针对性的扩展解决这一限制，将 Panda3D 打造成灵活、高吞吐量的批量渲染系统，适用于基于像素的强化学习。这些扩展包括：(1) 多视图平铺渲染，(2) 通过自定义着色器对实例化几何体进行基于张量的批量控制，(3) CUDA 与 OpenGL 互操作，以及 (4) 与 TorchRL 集成。下面将依次详细介绍这些组件及其作用。

多视图平铺该技术将 K 个相互独立的场景（例如，K 个并行强化学习环境）渲染到同一个大型离屏渲染目标中。该目标被划分为一个由互不重叠的图块组成的网格，并为每个场景分配各自的视图投影矩阵。顶点着色器中的轻量级视口重映射操作会将每个场景的几何体投影到指定图块中。此方法将 K 个渲染通道合并为一个，从而降低引擎和驱动程序的开销。

硬件实例化几何体控制 我们通过硬件实例化绕过 Python 应用程序接口来控制对象。PyBatchRender 提供了一个接口，用户可通过单条张量命令控制整批

彼此独立的对象；这类命令在 PyTorch 和 NumPy 中很常见。对于 S 个场景和I 个给定模型的实例，用户需提供用于表示变换的张量（例如，$S \times I \times 3$位置、$S \times I \times 3$方向、$S \times I \times 1$缩放）以及用于表示外观的张量（例如，$S \times I \times 4$颜色）。

我们的库会将用户提交的张量以向量化方式组合为 4×4 模型矩阵，并将其打包到缓冲纹理中，供自定义 GLSL 着色器使用。这种架构会针对每种独特的模型类型发起一次实例化绘制调用，使图形处理器能够并行渲染所有场景中的全部实例，且无需承担 Python 循环开销。这项技术还允许在不同场景间共享实例变换，在多个摄像机视图共享许多对象的情况下（例如多智能体强化学习），可减少带宽占用。

CUDA-OpenGL 互操作性 对于 NVIDIA 图形处理器，我们实现了直接的 CUDA–OpenGL 互操作路径。此流程会将最终的 OpenGL 输出纹理直接映射到 CUDA 内存空间。随后通过设备到设备拷贝，将像素传输到预先分配的 GPU 缓冲区，并通过 DLPack 协议将其提供给 PyTorch。整个操作无需任何“主机往返传输”（GPU → CPU→ GPU），从而消除了 PCIe 总线这一数据传输瓶颈。

为确保广泛适用，我们还提供了可移植的 CPU 路径。这种备用方法适用于任何硬件，包括 AMD GPU 和 Apple Silicon。它会将纹理标准地提取到系统内存中的连续缓冲区中，然后以 PyTorch 张量的形式返回。尽管受 PCIe 传输限制，这一路径仍能确保在非 NVIDIA 系统上完整支持开发和运行。

系统集成PyBatchRender 有意保持与物理引擎无关。它被设计为一个模块化渲染组件。该库提供了一个简单的接口，物理引擎可通过张量更新对象状态（位置、方向、缩放），而 PyBatchRender 则负责可视化。这种职责分离使研究人员能够独立更换物理后端或渲染配置。

为便于集成到强化学习训练流水线中，PyBatchRender 提供了一个 TorchRL 封装器，实现了标准的 Gymnasium 环境接口。此集成可无缝使用 TorchRL 的并行环境工具，这些工具支持多进程向量化，且无需将数据从图形处理器复制到 CPU 以进行进程间通信。

结果

我们进行了两项主要实验来评估 PyBatchRender 的性能。首先，通过累积性能提升分析，验证各项核心架构优化的性能贡献。其次，我们将 PyBatchRender 与其他最先进的高吞吐渲染库进行基准测试。

为验证我们设计选择的有效性，我们逐步将每项优化添加到基线渲染器中。所有测量均使用在 64×64 分辨率下渲染的 CartPoleBalance 环境，并在三种具有代表性的 GPU 架构上进行：Apple M2（代表现代统一内存笔记本电脑）、NVIDIA RTX 4090（高端消费级 GPU）和 NVIDIA A100（数据中心级 GPU）。这些平台在内存带宽、光栅化能力以及 CPU 与 GPU 之间的通信开销方面呈现出不同的性能特征。

图2展示了在朴素的 Panda3D 基线实现中逐步添加组件所带来的累积性能提升。我们的基线（朴素的 Panda3D 实现）达到了每秒 1.1k–3.3k 帧。M2 的基线性能最强，这可能得益于其统一内存架构，避免了 PCIe 总线延迟。A100 针对通用计算而非光栅化进行了优化，因此基线结果最低。

通过在 Python 中迭代管理彼此独立的场景来引入分块渲染，可立即带来性能提升。RTX 4090、A100 和 M2 的性能分别提升了 5.2×、6.5× 和 7.8× 倍。此阶段将渲染整合到单个帧缓冲区和渲染通道中，而不是使用 K 个独立通道，从而降低了渲染目标切换开销。场景管理和对象操作仍受 CPU 限制，进一步凸显了 M2 相较于独立 GPU 的统一内存优势。

在 NVIDIA 硬件上加入 CUDA-OpenGL 互操作性，可消除帧回读的 PCIe 瓶颈。这使 RTX 4090 和 A100 的吞吐量提升 1.35–1.65 倍，使其性能更接近 M2。更重要的是，消除 CPU-GPU 传输瓶颈后，后续优化便能充分利用 GPU 能力，不再受主机端带宽限制。

最显著的提升来自结合基于着色器的批量控制实现的硬件实例化。通过将实例操作从 Python 循环转移到 GPU 并行张量运算，M2 的性能提升了 5×，而 RTX 4090 和 A100 分别取得了显著的 107× 和 150× 加速。这表明渲染工作负载已成功从受 CPU 限制转变为受 GPU 限制。

最后，多进程为 A100 和 M2 带来的提升微乎其微，因为它们的 GPU 已处于饱和状态。不过，它使 RTX 4090 的速度额外提升了 1.38 倍，这表明其 CPU 端驱动开销仍是一个较小的瓶颈，可以通过并行化来缓解。

接下来，我们将 PyBatchRender 的性能与其他面向 Python 强化学习、提供高吞吐量渲染的热门库进行比较：Isaac Lab、Maniskill 和 Madrona MJX。我们使用 CartPoleBalance 环境，在一系列分辨率下进行像素观测。

所有基准测试均在 Nvidia RTX 4090 上运行。为进行清晰的、以图形处理器为重点的比较，PyBatchRender 以单进程模式运行（即不使用多进程优化）。竞品库的性能数据取自（Zakka等，2025）。

图3所示的结果表明，在所有测试分辨率下，PyBatchRender 的性能都显著领先于所有参与比较的库。Isaac Lab 和 Maniskill 的性能表现相近，

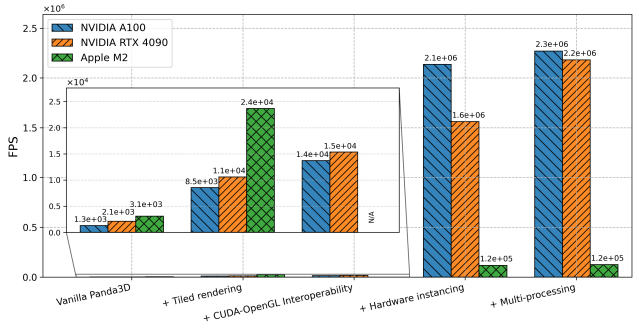

图2：在 CartPoleBalance 环境中以 64x64 分辨率进行渲染时，各项优化带来的累计性能提升。每项优化都在基准方案的基础上逐步叠加，展示了它们对 Apple M2、NVIDIA RTX 4090 和 NVIDIA A100 图形处理器最终性能的贡献。

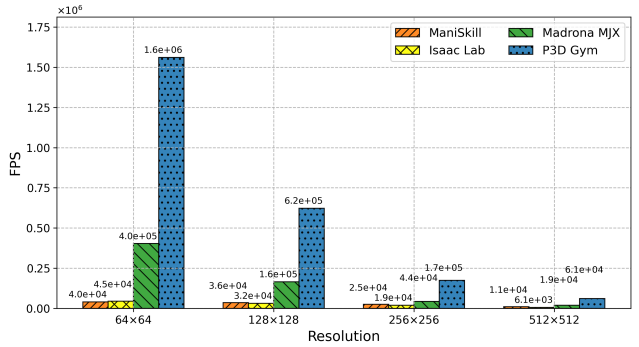

图3：在 NVIDIA RTX 4090 上，对比 CartpoleBalance 环境中带像素观测的环境渲染吞吐量。PyBatchRender（不使用多进程）与 Isaac Lab、Maniskill 和 Madrona MJX 在不同分辨率下进行比较。竞品数据引自（Zakka 等，2025年）。

其中 Maniskill 在较高分辨率下表现更好。PyBatchRender 的性能显著优于最接近的竞争者 Madrona MJX。在 64×64 下，PyBatchRender 达到每秒 160 万帧，而 Madrona MJX 为每秒 40.3 万帧。这种差距在不同分辨率下始终存在，只是略有缩小；在 512×512 下，PyBatchRender 的渲染速度为每秒 6.1 万帧，而 Madrona MJX 为每秒 1.9 万帧（快 3.2× 倍）。

讨论

PyBatchRender 在保持纯 Python 易用性的同时，实现了超过 100 万每秒帧数的性能，证明高吞吐渲染无需从零开始构建专用引擎。通过针对性优化扩展现有的成熟框架，我们以显著降低的复杂度实现了具有竞争力的性能。不过，仍需结合渲染技术的权衡以及实际训练流水线的限制来审视这些结果。

PyBatchRender 基于光栅化的架构体现了经过深思熟虑的设计选择，也带来了重要的权衡。正如 Rosenzweig等，2024 所示，当以下三个条件同时满足时，光线追踪的性能可能优于光栅化：低分辨率、高多边形数量，以及专用于光栅化的核心数量较少的数据中心 GPU。不过，这种性能反超的适用场景在实际中意义有限。低分辨率观测无法呈现高多边形网格所提供的几何细节。另一种方法是采用细节层次技术，根据分辨率相应降低网格复杂度；在这种情况下，光栅化仍然极具竞争力。此外，随着分辨率提高并接近 128×128–256×256 这一现代视觉强化学习中日益常见的水平，光线追踪的优势也会减弱（Zhou等，2025；Kim、Finn和Liang，2025）。

尽管 PyBatchRender 的渲染速度超过每秒 100 万帧，但必须结合完整的训练流水线来评估渲染性能。在 RTX 4090 上，经过优化的近端策略优化（PPO）实现每秒可处理约 3 万至 5 万帧（Zakka 等，2025年）。渲染速度一旦超过这一阈值，渲染便不再是主要瓶颈；继续提升渲染速度，对端到端训练时间的改善也会逐渐减小。这一观察为我们的设计理念提供了依据：性能达到足够水平（每秒 5 万至 10 万帧）后，与其追求极高吞吐量，不如将精力投入到简洁的接口、丰富的功能和生态系统集成上。

近期将为 PyBatchRender 环境实现若干扩展功能，包括原生支持动画骨骼网格、动态阴影映射和高级逐实例剔除。尽管 Panda3D 提供了这些功能，但直接集成会重新引入逐对象开销，导致性能下降。目前，该库面向 PyTorch 开发，发挥其灵活性和主流生态的优势，从而支持与 TorchRL 无缝集成及高效的多进程并行。不过，JAX 生态凭借 Brax、MuJoCo MJX 和基于 JAX 的训练实现，在强化学习领域日益受到关注。未来的工作应考虑支持 JAX，以便与这些物理模拟器实现更紧密的集成，并有望利用即时编译降低环境步进的开销。

PyBatchRender 背后的核心动机不止于原始性能指标。通过将高吞吐量与 Python 的易用性相结合，我们旨在让基于像素的强化学习普及开来。过去，要在基于像素的强化学习中取得有竞争力的性能，要么需要具备系统编程专长，要么只能接受引擎封装架构带来的通信瓶颈。PyBatchRender 的性能可与专用的 C++ 系统相媲美，同时仍能保持快速原型开发周期，因为研究人员只需用 30–100 行纯 Python 代码就能定义自定义环境。与物理引擎无关的设计进一步支持多样化的研究方向，使团队能够采用高性能渲染，而不必绑定特定的物理后端。这种模块化和易用性使规模较小的研究团队和个人研究人员也能开展大规模视觉强化学习实验，而过去这类实验实际上只有资源充足的实验室才能完成。

参考文献

Beeching, E.; Debangoye, J.; Simonin, O.; and Wolf, C. 2021. Godot强化学习智能体. arXiv:2112.03636 [cs]. Berges, V.-P.; Szot, A.; Chaplot, D. S.; Gokaslan, A.; Mottaghi, R.; Batra, D.; and Undersander, E. 2023. Galactic：以每秒 10 万步的速度扩展用于重排任务的端到端强化学习. arXiv:2306.07552 [cs]. Goslin, M.; and Mine, M. 2004. Panda3D 图形引擎. 计算机, 37(10): 112–114. Kiefl, E. 2024. Pooltool：用于逼真台球仿真的 Python 软件包. 开放源代码软件期刊, 9(101): 7301. Kim, M. J.; Finn, C.; and Liang, P. 2025年. 微调视觉-语言-动作模型：优化速度与成功率. arXiv:2502.19645 [cs]. Li, Q.; Peng, Z.; Feng, L.; Zhang, Q.; Xue, Z.; and Zhou, B. 2023. MetaDrive：组合多样化驾驶场景，实现可泛化的强化学习. *IEEE*模式分析与机器智能汇刊, 45(3): 3461–3475. Mittal, M.; Yu, C.; Yu, Q.; Liu, J.; Rudin, N.; Hoeller, D.; Yuan, J. L.; Singh, R.; Guo, Y.; Mazhar, H.; Mandlekar, A.; Babich, B.; State, G.; Hutter, M.; and Garg, A. 2023. Orbit：面向交互式机器人学习环境的统一仿真框架. *IEEE*机器人与自动化快报, 8(6): 3740–3747. arXiv:2301.04195 [cs]. Mu, T.; Ling, Z.; Xiang, F.; Yang, D.; Li, X.; Tao, S.; Huang, Z.; Jia, Z.; and Su, H. 2021. ManiSkill：基于大规模演示的可泛化操作技能基准. arXiv:2107.14483 [cs]. Rosenzweig, L. G.; Shacklett, B.; Xia, W.; and Fatahalian, K. 2024. 面向具身人工智能的高吞吐量批量渲染. 收录于 *SIGGRAPH Asia 2024* 会议论文集, SA '24, 1–9. 美国纽约州纽约：计算机协会。ISBN 979-8-4007-1131-2. Shacklett, B.; Rosenzweig, L. G.; Xie, Z.; Sarkar, B.; Szot, A.; Wijmans, E.; Koltun, V.; Batra, D.; and Fatahalian, K. 2023. 面向高性能多世界仿真的可扩展面向数据架构. *ACM* 图形学汇刊, 42(4): 90:1–90:13. Shacklett, B.; Wijmans, E.; Petrenko, A.; Savva, M.; Batra, D.; Koltun, V.; and Fatahalian, K. 2021. 面向深度强化学习的大批量仿真. arXiv:2103.07013[cs]. Tao, S.; Xiang, F.; Shukla, A.; Qin, Y.; Hinrichsen, X.; Yuan, X.; Bao, C.; Lin, X.; Liu, Y.; Chan, T.-k.; Gao, Y.; Li, X.; Mu, T.; Xiao, N.; Gurha, A.; Rajesh, V. N.; Choi, Y. W.; Chen, Y.-R.; Huang, Z.; Calandra, R.; Chen, R.; Luo, S.; and Su, H. 2025年. ManiSkill3：面向可泛化具身人工智能的图形处理器并行机器人仿真与渲染. arXiv:2410.00425 [cs]. Unity Technologies. 2025年. Unity-Technologies/ml-agents. 原始日期：2017-09-08T21:09:04Z. Yarats, D.; Zhang, A.; Kostrikov, I.; Amos, B.; Pineau, J.; and Fergus, R. 2019. 提升基于图像的无模型强化学习的样本效率。Zakka, K.; Tabanpour, B.; Liao, Q.; Haiderbhai, M.; Holt, S.; Luo, J. Y.; Allshire, A.; Frey, E.; Sreenath, K.; Kahrs, L. A.; Sferrazza, C.; Tassa, Y.; and Abbeel,P. 2025年。MuJoCo Playground。arXiv:2502.08844 [cs]。Zhou, Z.; Atreya, P.; Tan, Y. L.; Pertsch, K.; and Levine, S. 2025年。AutoEval：真实世界中通用机器人操作策略的自主评估。载于第九届机器人学习会议论文集，1997–2017。PMLR。国际标准连续出版物号：2640-3498。