DUMPSARENA

OpenUSD Development

NVIDIA NCP-OUSD

Version Demo

Total Demo Questions: 8

Total Premium Questions: 85

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, OpenUSD Fundamentals	3
Topic 2, OpenUSD Composition	33
Topic 3, OpenUSD Data Modeling	25
Topic 4, OpenUSD APIs	13
Topic 5, OpenUSD Ecosystem	10
Topic 6, Mix Questions	1
Total	85

QUESTION NO: 1

You are a developer creating an OpenUSD exporter for an application that also supports import of USD assets. To enable collaborative workflows, you're adding an â€œexport as overridesâ€□ option.

Which approach correctly describes which structure your exporter should generate?

A. Overs of the prims and properties that have been modified or added, omitting unchanged data.

B. Export each prim separately into multiple layers, and reference them individually to maintain sparsity.

C. Include explicit definitions of all prims, properties, and relationships exactly matching the imported asset to ensure consistency.

ANSWER: A

Explanation:

Overs of the prims and properties that have been modified or added, omitting unchanged data. is the appropriate structure for an export-as-overrides workflow. In OpenUSD, an over is a prim spec that contributes opinions to a prim without redefining its complete underlying structure. A stronger layer containing overs can therefore modify values, add properties, establish relationships, or introduce new prims while relying on composed weaker layers for all source content that has not changed.

This keeps the exported layer sparse: it captures the unique contribution from the current application or workstream rather than making a duplicate of the imported asset. Such sparse authoring is essential for round-trip and multi-workstream exchange because independently authored layers can compose non-destructively. It also makes review, version control, and downstream updates more manageable, since the layer clearly represents the intended delta. Where new scene content is created, the exporter can author the required defining specs for that new content; for pre-existing composed content, it should author only the changed opinions as overs.

This behavior follows the OpenUSD composition model described in the OpenUSD over glossary entry and NVIDIA's OpenUSD asset-structure guidance.

QUESTION NO: 2

Which of these is a viable approach for mapping or grouping compound types from other data sources to OpenUSD?

A. arrays

B. namespace-prefixed attributes

C. structs

ANSWER: B

Explanation:

Namespace-prefixed attributes are a viable way to represent compound or grouped data when bringing information from another data source into OpenUSD. USD properties are named within a namespace, using colon-separated identifiers such as `source:group:member`. This lets an adapter retain the logical grouping and member names of an external compound value while storing each member as an individual, independently typed attribute on a prim. For example, an imported data source with a grouped set of settings can be represented by attributes such as `settings:render:quality` and `settings:render:samples`. Consumers can then discover or query the entire related property set through USD namespace APIs, while still reading, authoring, time-sampling, and composing each value using normal attribute behavior. This approach aligns with USD's property model and preserves useful semantic organization without requiring a custom aggregate value type. The OpenUSD API documentation describes how property names use namespaces and provides utilities for working with namespace prefixes, including grouping and querying related property names. See the SdfPath API reference and the UsdPrim API reference.

QUESTION NO: 3

Consider the following prim hierarchies. Each prim hierarchy is represented by nested unordered lists and each list item represents a prim with the following format: PRIM_NAME (KIND). No text within the parentheses means that the kind is unset for that prim. Only evaluate model kind hierarchy correctness and the structural choices. Which prim hierarchies represent valid model kind hierarchies? Choose three.

A. CarA (component)Interior ()SteeringWheel ()Chasis ()Wheels ()Wheel_FR ()Wheel_RL ()

B. CarA (assembly)Interior (group)SteeringWheel (component)Chasis ()Wheels (group)Wheel_FR (component)Wheel_RL (component)

C. CarA (component)Interior ()SteeringWheel (component)Chasis ()Wheels ()Wheel_FR (component)Wheel_RL (component)

D. CarA (component)Interior ()SteeringWheel (subcomponent)Chasis ()Wheels (group)Wheel_FR (subcomponent)Wheel_RL (subcomponent)

E. CarA (component)Interior ()SteeringWheel (subcomponent)Chasis ()Wheels ()Wheel_FR (subcomponent)Wheel_RL (subcomponent)

ANSWER: A B E

Explanation:

The valid hierarchies are the one headed by *CarA (component)* with all descendant kinds unset, the one headed by *CarA (assembly)* that uses *Interior (group)* and *Wheels (group)* to organize component models, and the one headed by *CarA (component)* that contains *SteeringWheel (subcomponent)* plus subcomponent wheel prims.

In OpenUSD, `group`, `assembly`, and `component` participate in the model hierarchy. A group is an organizational model, and assembly is a specialized group that can contain other models. This makes the hierarchy with the assembly root, group-level Interior and Wheels containers, and component-level leaf assets structurally valid. A component represents a leaf asset model and establishes a pruning boundary for nested model kinds. It may still contain ordinary prims with no kind authored, so the component-root hierarchy whose descendants are unset is valid. It may also contain `subcomponent` prims for semantically important internal parts. Subcomponent is intentionally not a model kind, so those internal annotations do not introduce nested models below the component boundary. These relationships follow NVIDIA's model-kind hierarchy guidance and the OpenUSD Kind registry documentation: NVIDIA Learn OpenUSD: Model Kinds and OpenUSD Kind Registry.

QUESTION NO: 4

What is the only reliable way in OpenUSD of encoding the motion of primitives whose topology is varying over time?

A. positions

B. velocities

C. orientations

ANSWER: B

Explanation:

velocities is correct because OpenUSD uses velocity data to represent motion safely when a primitive's topology changes between time samples. Topology-varying geometry can gain, lose, reorder, or otherwise change its points over time. Consequently, the array elements at one sample cannot reliably be assumed to correspond to elements at a neighboring sample. Motion cannot therefore be dependably reconstructed by interpolating point positions across those samples.

A velocity is associated with a point at the relevant sample and supplies its instantaneous direction and rate of movement. OpenUSD's geometry schemas can use this information to compute motion-blurred positions over a shutter interval without requiring point arrays at adjacent samples to have identical counts or stable index correspondence. This is especially important for effects such as particles, fluid meshes, or other generated geometry whose component structure evolves frame

by frame. The OpenUSD `UsdGeomPointBased` API documentation explicitly states that using velocities is the only reliable way to encode motion for primitives with varying topology. See the UsdGeomPointBased API reference and the OpenUSD geometry documentation.

QUESTION NO: 5

When designing a scalable asset structure, which two aspects should be primarily considered? Choose two.

A. Innovation and future-proofing

B. File formats and compression methods

C. Clients and collaborators

D. Modularity and performance

ANSWER: C D

Explanation:

Clients and collaborators should be a primary consideration because an OpenUSD asset structure exists to support the people, departments, tools, and downstream workflows that create, assemble, review, and deliver content. A scalable design is therefore tailored to the consumers of the asset and to the collaboration model required by the production. OpenUSD is designed for composition and non-destructive collaboration, so asset organization should enable contributors to work independently where appropriate while providing reliable integration for the assembled result.

Modularity and performance are also primary design considerations. Modularity lets an asset be divided into reusable, maintainable components that can be composed in different contexts, supporting parallel authoring and reducing unnecessary duplication. Performance ensures that the resulting structure remains practical at scale: composition, loading, traversal, and interaction must remain efficient as asset complexity and the number of collaborating users grow. NVIDIA's OpenUSD learning material identifies modularity and performance among the key principles for scalable asset structures, alongside legibility and navigability. These principles help teams establish structures that are usable both immediately and throughout the life of a production. See NVIDIA's Asset Structure guidance and the OpenUSD introduction.

QUESTION NO: 6

How does the concept of an edit target (Usd.EditTarget) interact with the stage in OpenUSD?

A. It merges edits across all layers automatically for simplified editing of a stage.

B. It overrides all layers, forcing every change to be written to the root layer.

C. It specifies the destination layer for authoring changes in a composed stage.

D. It temporarily disables all sublayer compositions during editing.

ANSWER: C

Explanation:

An edit target specifies where edits made through a `UsdStage` are authored. A stage presents a composed view of scene data that can draw opinions from a root layer, sublayers, references, payloads, variants, and other composition arcs. Because this composed result can involve many contributing layers, OpenUSD needs an explicit target that determines the layer (and, when applicable, the mapping into that layer's namespace) that receives new authored opinions. The stage has a current edit target, available through `GetEditTarget()`, and an application can change it with `SetEditTarget()`. Once selected, operations such as setting an attribute value, creating a prim, or changing metadata author their corresponding scene-description changes into the edit target rather than directly modifying the composed result itself. This separation is essential to OpenUSD's non-destructive workflow: users can place overrides into a dedicated stronger layer while preserving source asset layers and the composition structure. The target must be valid for the stage's composition and its layer must be editable for authoring to succeed. See the OpenUSD API documentation for UsdStage and UsdEditTarget.

QUESTION NO: 7

A shot stage references a published asset and has a stronger editable override layer in its local layer stack. An artist must author a new translate value for the referenced asset without changing the published asset layer. Which statements about edit targets are correct? Choose two.

A. The edit target determines where a USD authoring operation writes its opinion.

B. Setting the override layer as the edit target supports a non-destructive local transform override.

C. An edit target writes changes directly into the composed stage rather than into a layer.

D. Setting any edit target forces subsequent changes to be authored in the root layer.

ANSWER: A B

Explanation:

The stage's current `UsdEditTarget` determines the layer and mapping into which USD authoring operations write opinions. Setting the editable override layer as the edit target allows the artist to author a stronger local transform opinion while preserving the published asset layer.

An edit target does not edit the composed result directly. The composed stage is a resolved view of opinions authored in contributing layers. Nor does setting an edit target automatically make the root layer the destination; the target can be changed to an appropriate editable layer in the stage's local layer stack.

QUESTION NO: 8

Which of the following sentences are correct when converting between .usda, .usdc, .usd and .usdz files? Choose two.

A. A .usda file can be converted to a .usdz file using usdcat.

B. A .usda file can be converted to a .usdc file using usdcat.

C. A .usda file can be converted to a .usd file by simply renaming it.

D. A .usda file can be converted to a .usdc file by simply renaming it.

ANSWER: B C

Explanation:

“A .usda file can be converted to a .usdc file using usdcat.” is correct because `usdcat` reads a USD layer and writes it using the layer format inferred from the output filename extension. Therefore, specifying a `.usdc` output file causes the textual USDA layer to be written in USD’s binary crate format. For example, the OpenUSD conversion tutorial shows the pattern `usdcat -o NewSphere.usdc Sphere.usda`. This is a genuine file-format conversion, rather than a change to the filename alone.

“A .usda file can be converted to a .usd file by simply renaming it.” is also correct. The `.usd` extension is intentionally format-neutral: it may contain either text USDA data or binary USDC data. Consequently, changing a USDA file’s extension to `.usd` does not need to rewrite its contents. When opening the asset, the USD runtime identifies the underlying layer format from the file content and selects the appropriate format plugin. This convention is useful when a pipeline wants a generic USD filename while retaining the existing text or crate representation. See the official OpenUSD tutorial on converting between layer formats and the SdfFileFormat API documentation.