Bringing Photoshop to the iPad

Sean Parent | Senior Principal Scientist

#AdobeRemix
Vasjen Katro / Baugasm

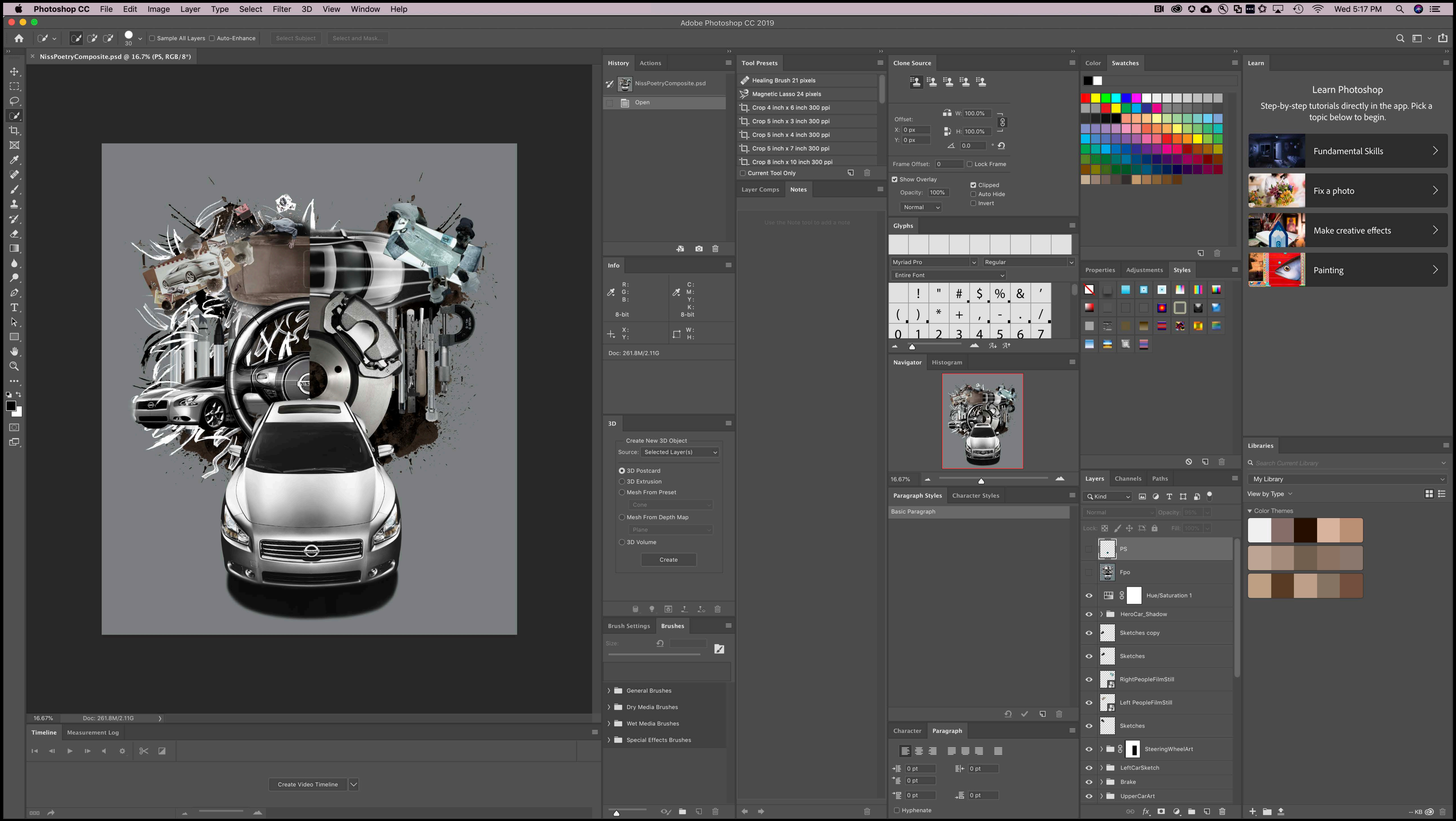
photoshopishiring.com

"Could we just put Photoshop on the iPad?"

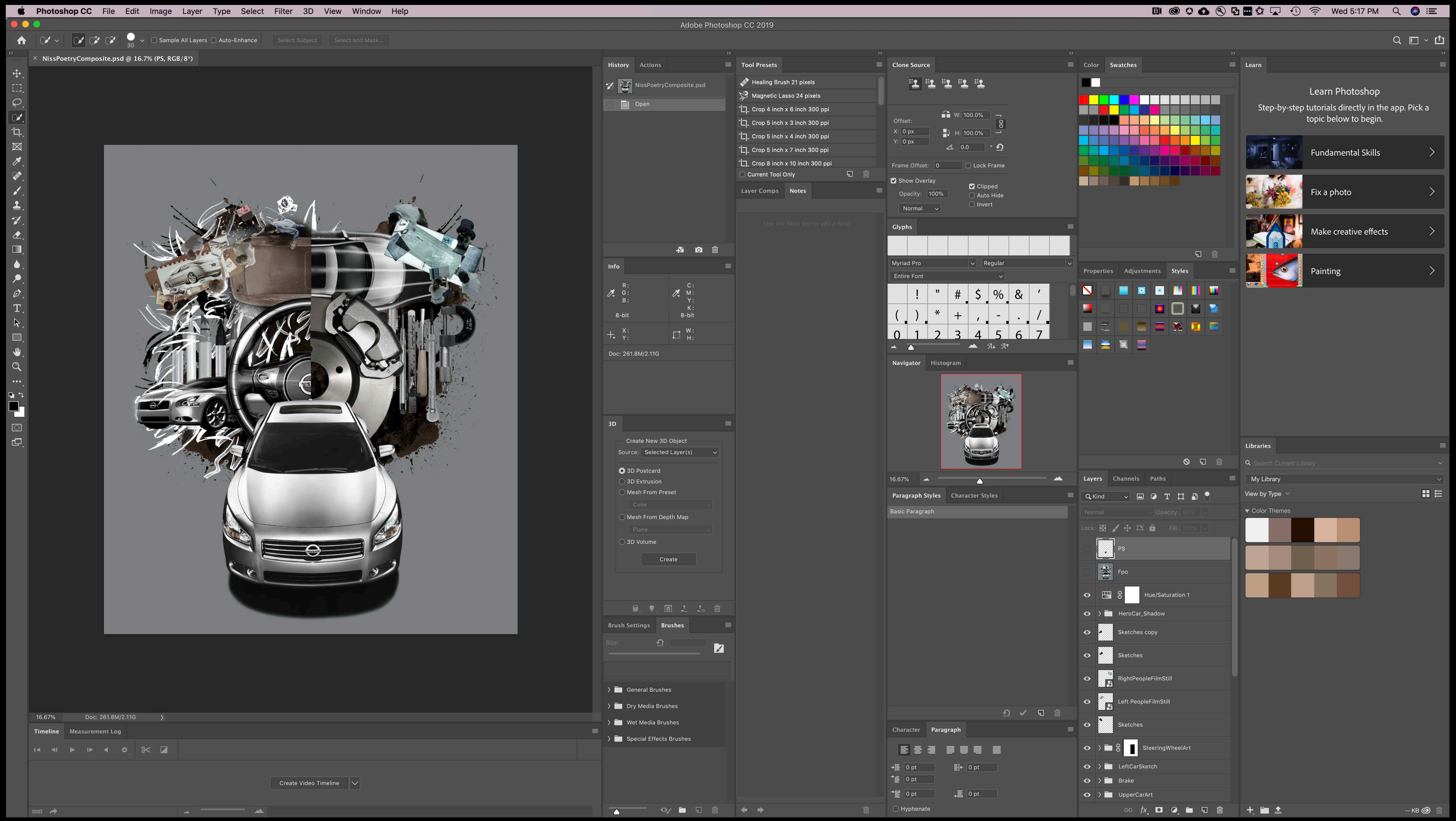
– David Howe

Adobe

Demo

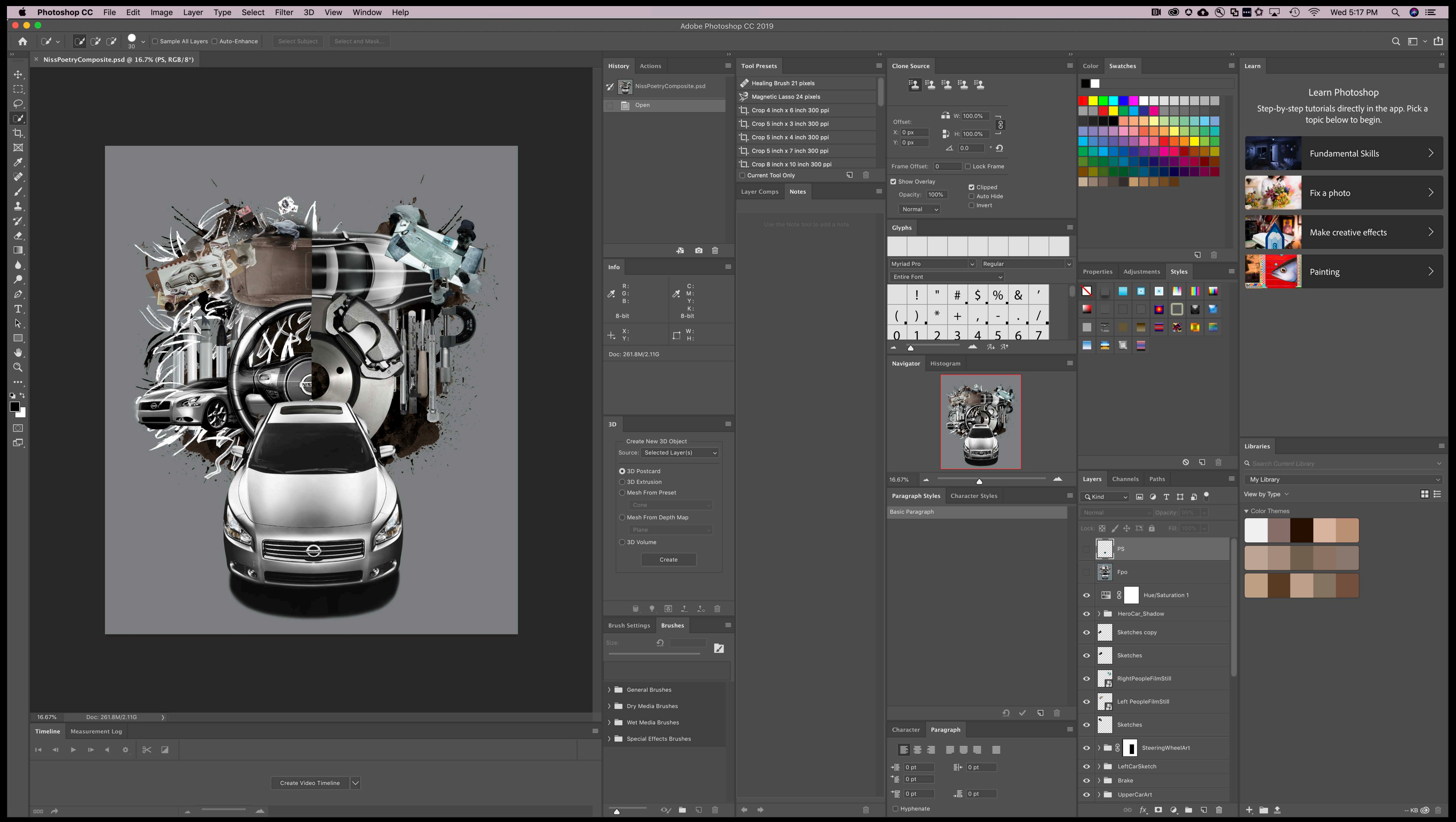

Photoshop CC File Edit Image Layer Type Select Filter 3D View Window Help
Wed 5:17 PM
Adobe Photoshop CC 2019
Sample All Layers
Auto-Enhance
Select Subject
Select and Mask...
NissPoetryComposite.psd @ 16.7% (PS, RGB/8*)
History
Actions
Open
Tool Presets
Healing Brush 21 pixels
Magnetic Lasso 24 pixels
Crop 4 inch x 6 inch 300 ppi
Crop 5 inch x 3 inch 300 ppi
Crop 5 inch x 4 inch 300 ppi
Crop 5 inch x 7 inch 300 ppi
Crop 8 inch x 10 inch 300 ppi
Current Tool Only
Clone Source
Color
Swatches
Learn
Learn Photoshop
Step-by-step tutorials directly in the app. Pick a topic below to begin.
Fundamental Skills
Fix a photo
Make creative effects
Painting
Layer Comps
Notes
Glyphs
Info
Doc: 261.8M/2.11G
Properties
Adjustments
Styles
Navigator
Histogram
16.67%
3D
Create New 3D Object
3D Postcard
3D Extrusion
Mesh From Preset
Mesh From Depth Map
3D Volume
Create
Paragraph Styles
Character Styles
Basic Paragraph
Layers
Channels
Paths
Libraries
My Library
View by Type
Color Themes
PS
Fpo
Hue/Saturation 1
HeroCar_Shadow
Sketches copy
Sketches
RightPeopleFilmStill
Left PeopleFilmStill
Sketches
SteeringWheelArt
LeftCarSketch
Brake
UpperCarArt
Brush Settings
Brushes
General Brushes
Dry Media Brushes
Wet Media Brushes
Special Effects Brushes
Timeline
Measurement Log
Create Video Timeline
Character
Paragraph
Hyphenate

Challenge: Photoshop is Complex



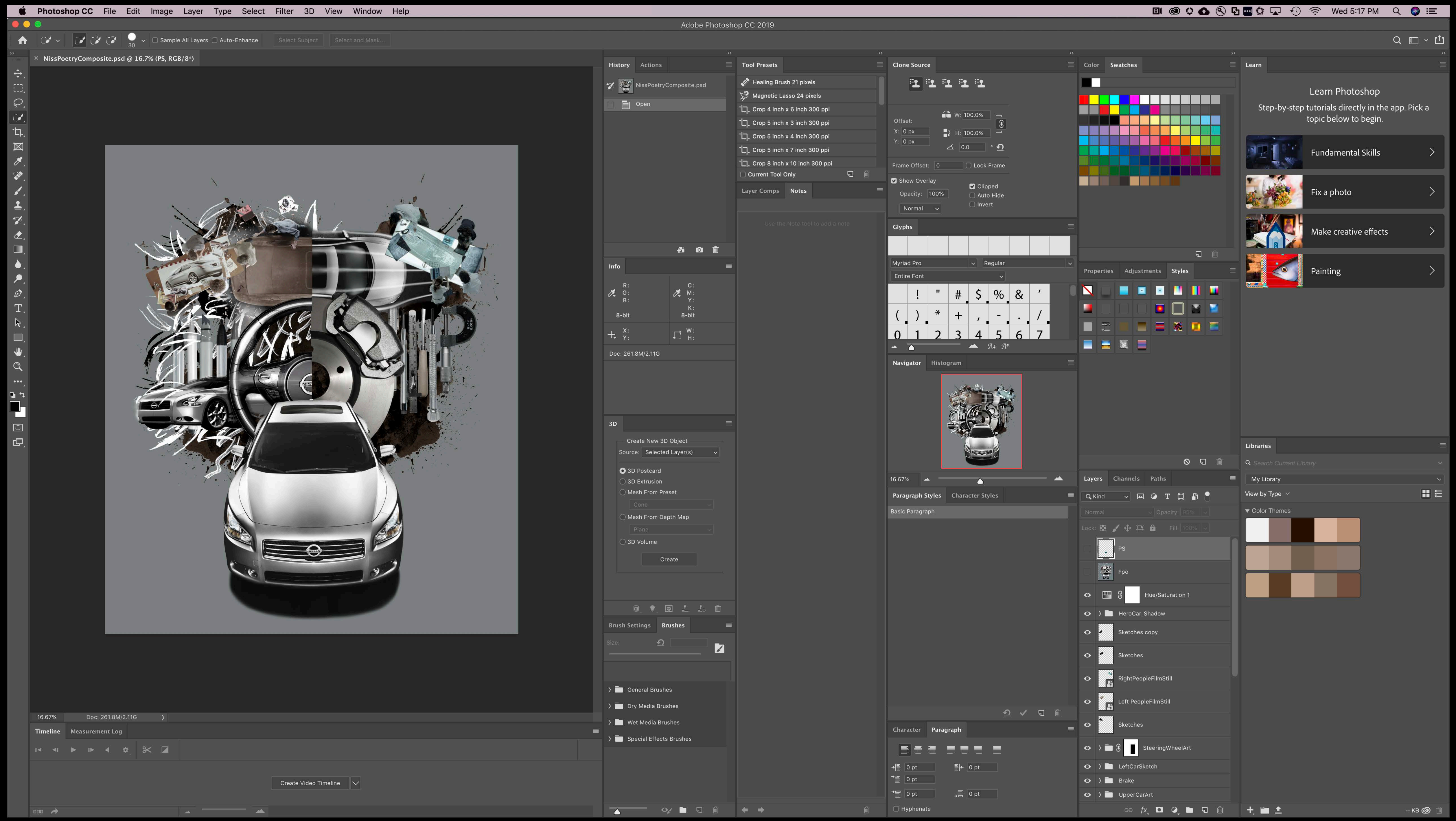
Challenge: Photoshop is Complex

- There is no explicit model



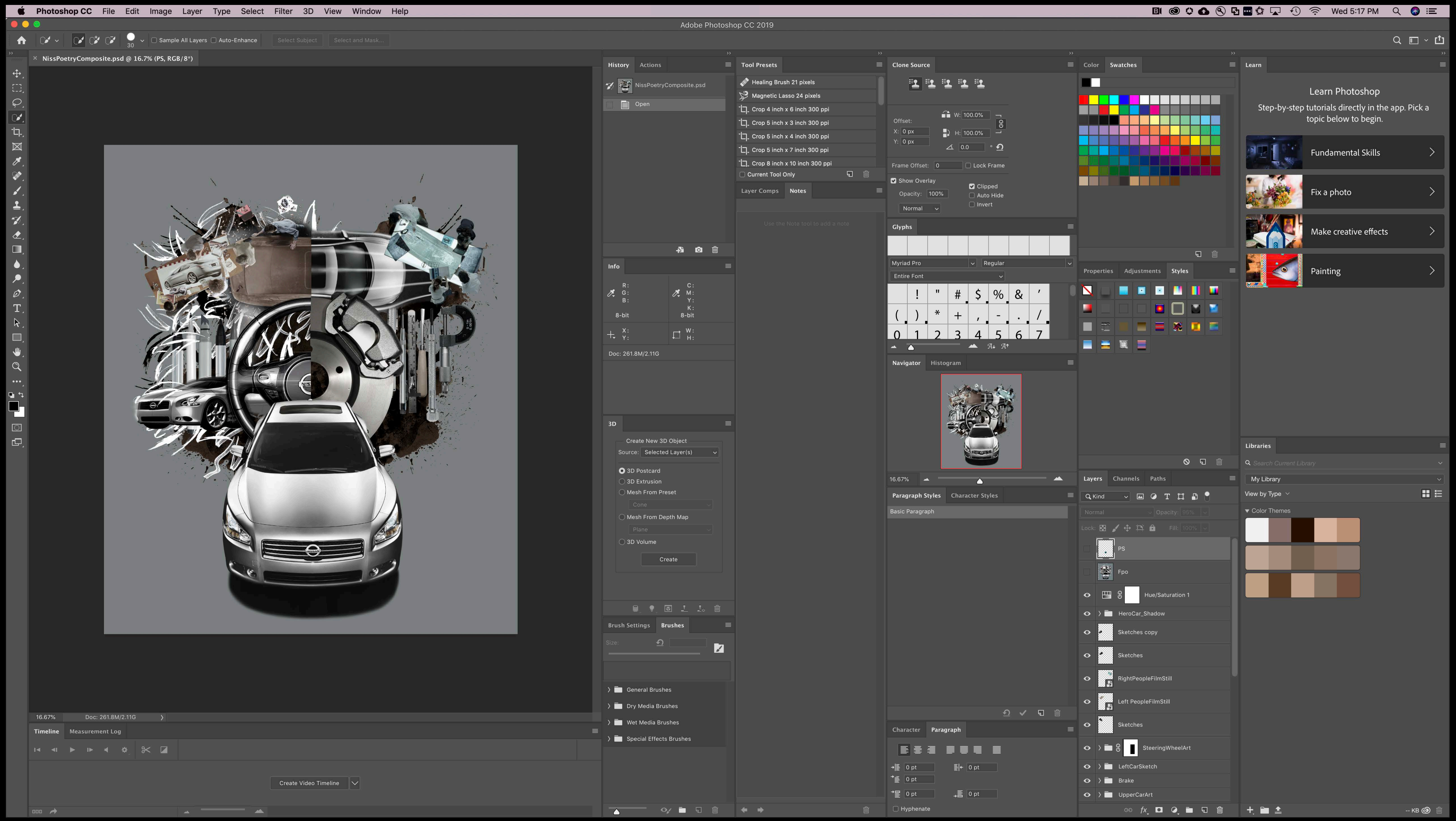
Challenge: Photoshop is Complex

- There is no explicit model
  - A model in MVC is:



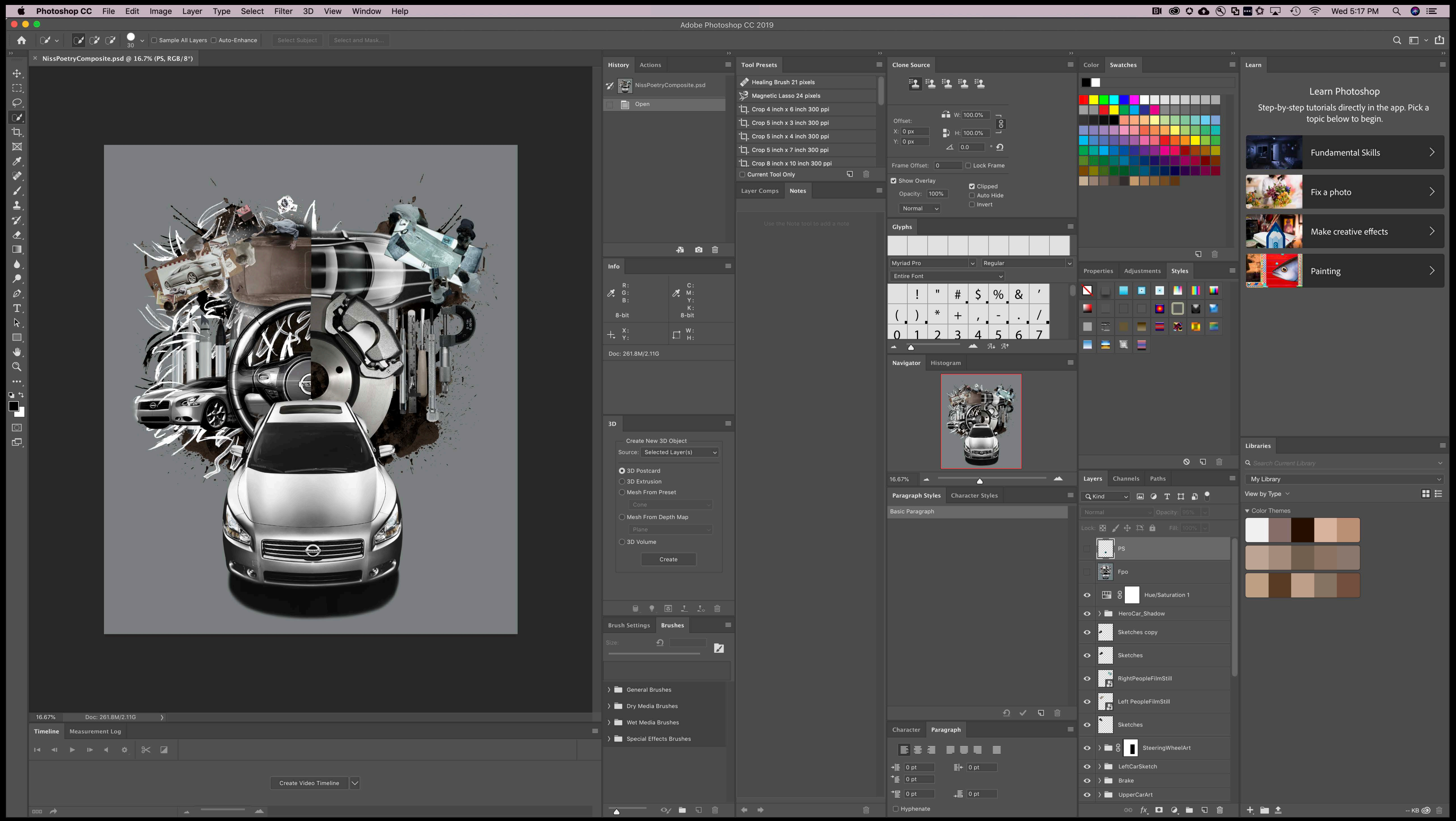
Challenge: Photoshop is Complex

- There is no explicit model
  - A model in MVC is:
    - Data



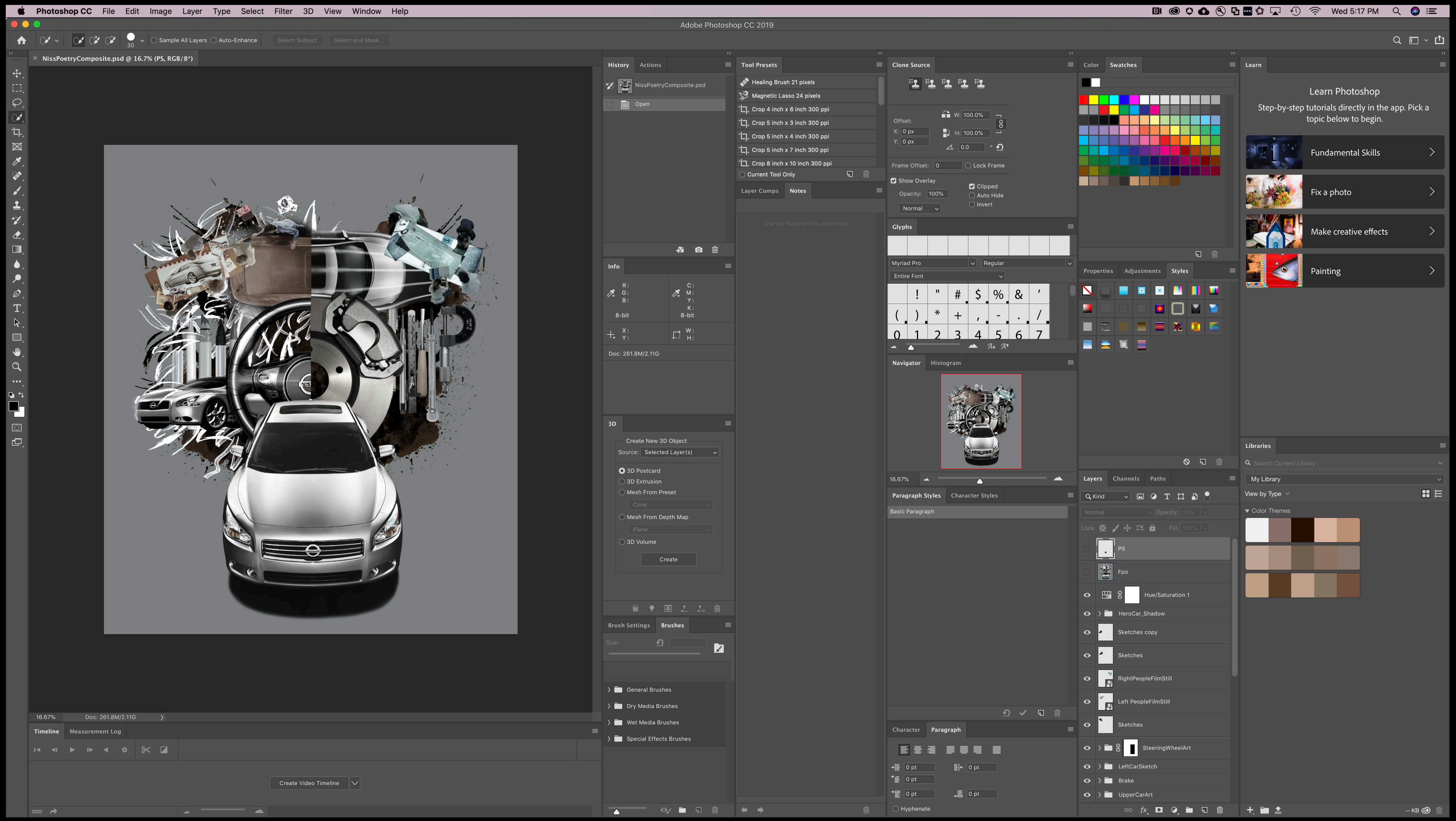
Challenge: Photoshop is Complex

- There is no explicit model
  - A model in MVC is:
    - Data
    - Constraints and Relationships



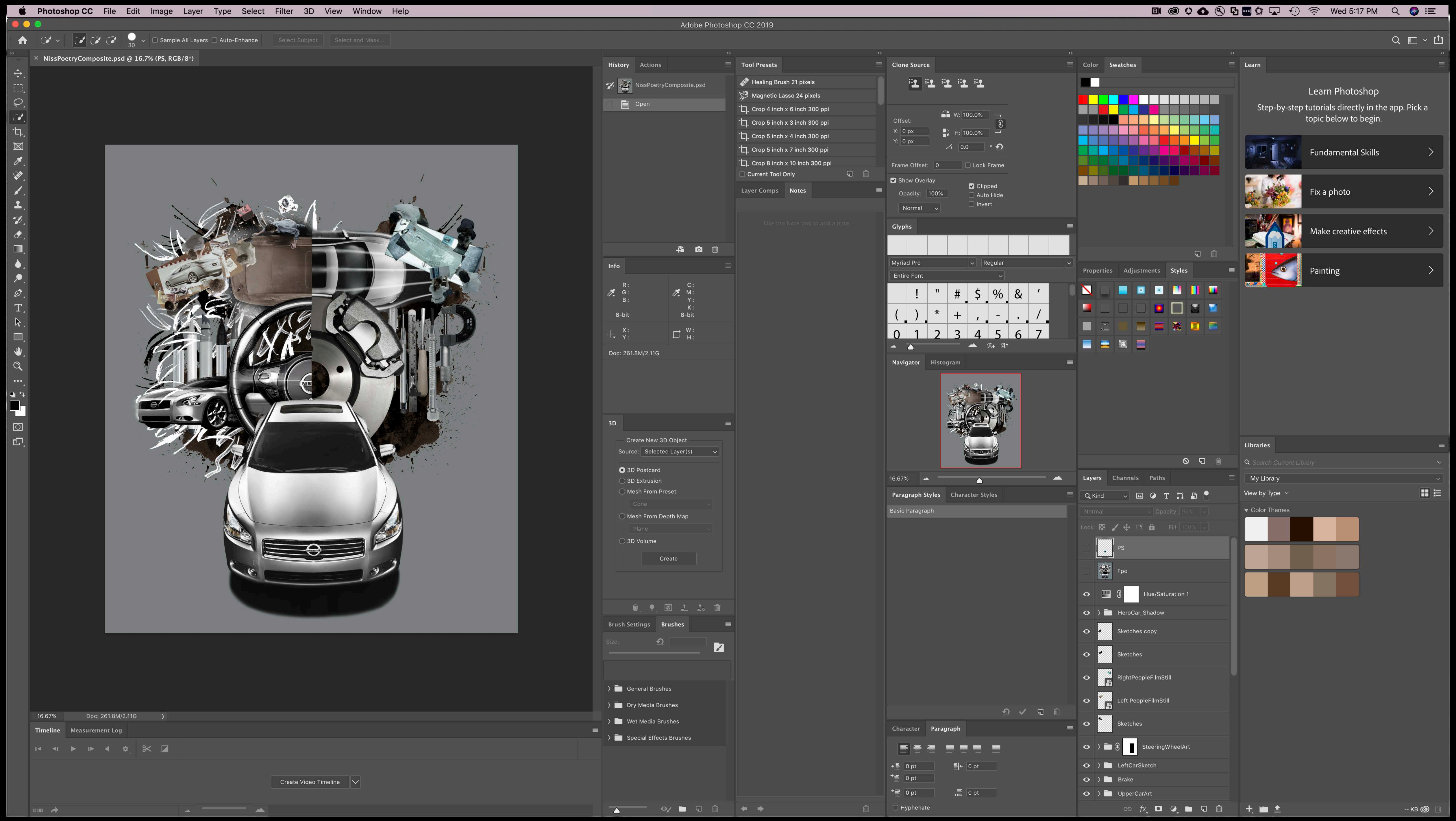
Challenge: Photoshop is Complex

- There is no explicit model
  - A model in MVC is:
    - Data
    - Constraints and Relationships
    - Observable



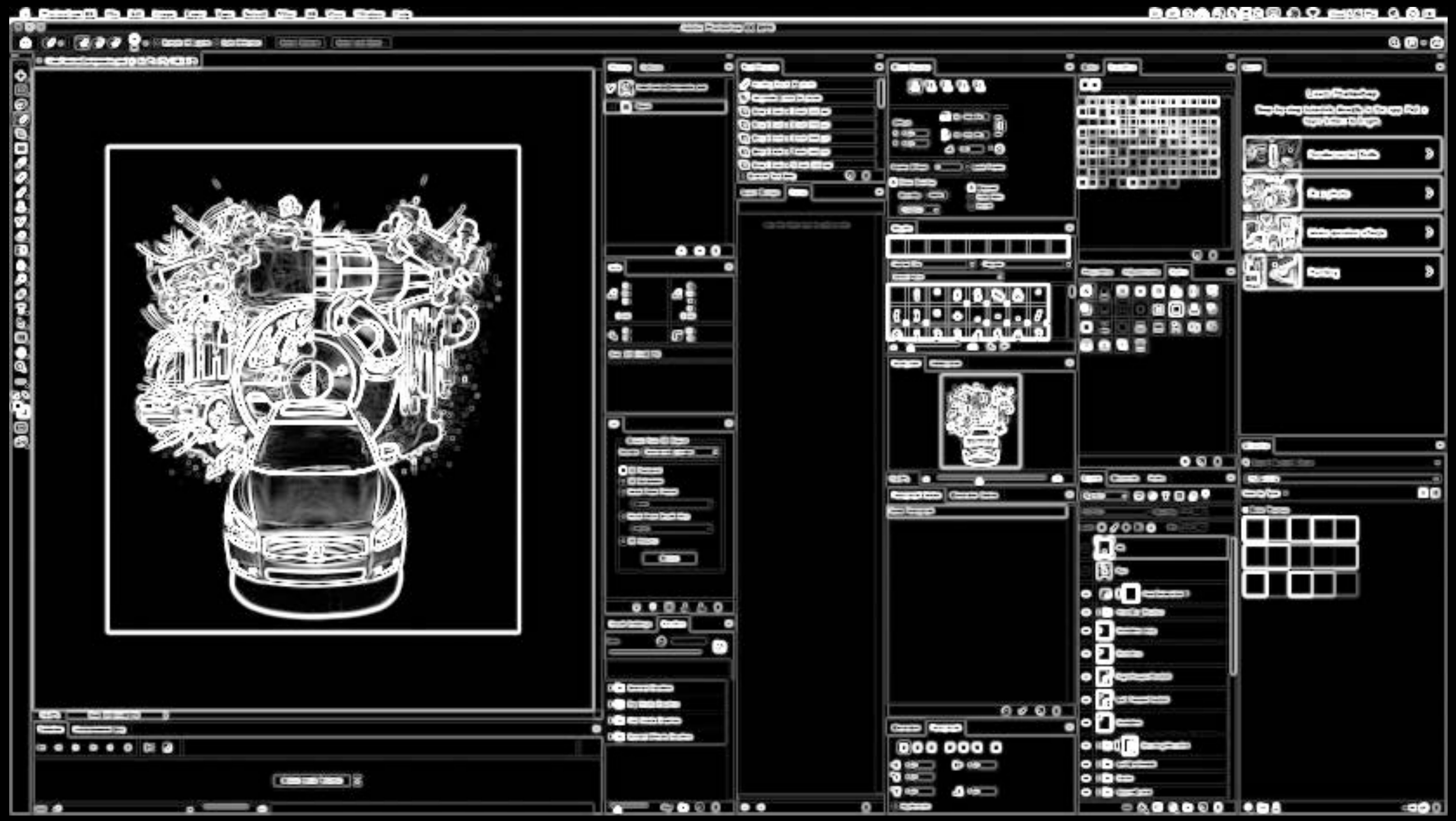
The UI *is* the Model

Solution


Photoshop CC File Edit Image Layer Type Select Filter 3D View Window Help
Adobe Photoshop CC 2019
Learn Photoshop
Step-by-step tutorials directly in the app. Pick a topic below to begin.
Fundamental Skills
Fix a photo
Make creative effects
Painting

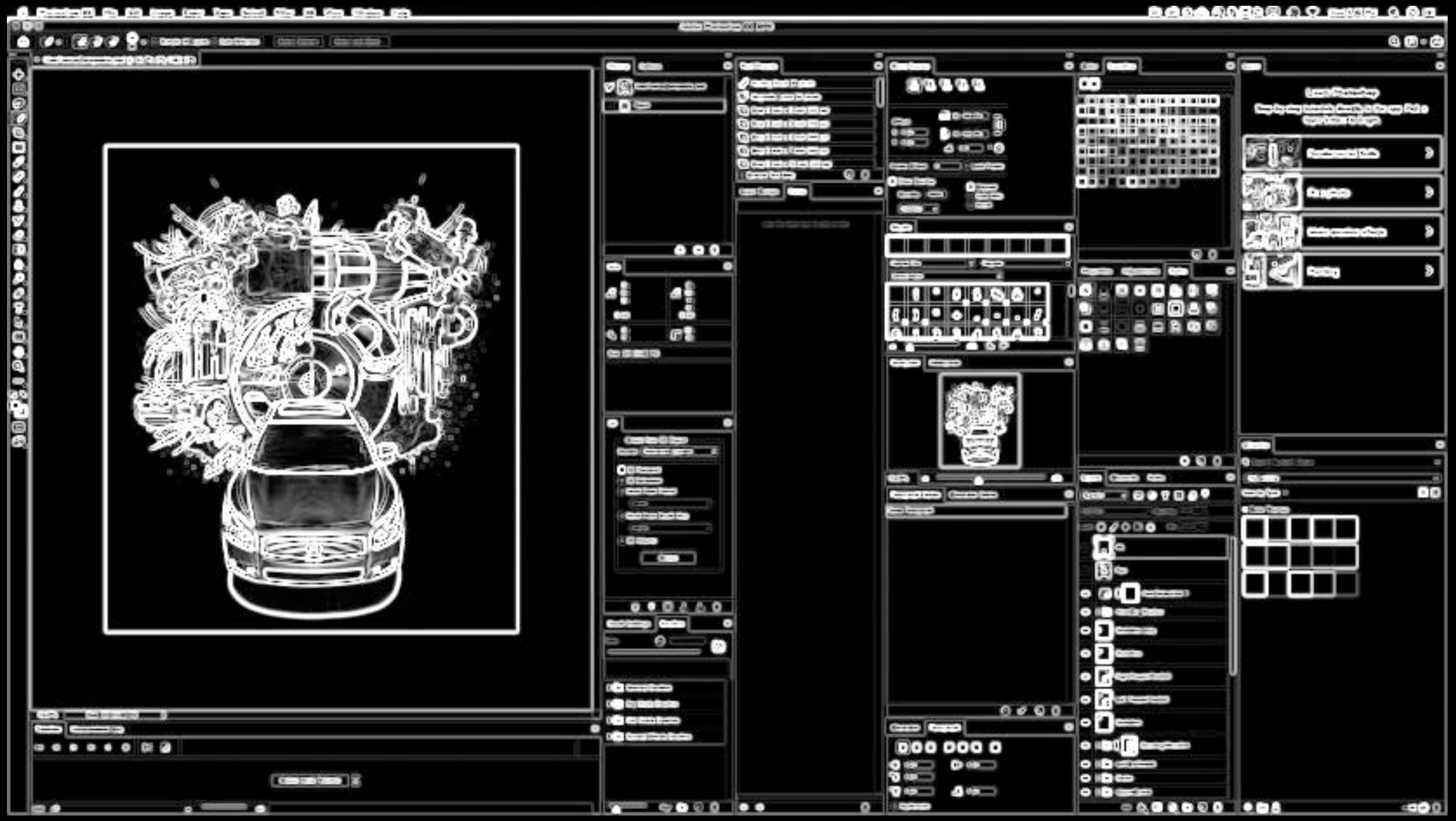
Solution

- Remove the visual representation from the UI elements



Solution

- Remove the visual representation from the UI elements
- Add the ability to bind to, to *observe*, UI elements



Solution

- Remove the visual representation from the UI elements
- Add the ability to bind to, to *observe*, UI elements
- Lazily instantiate UI elements as needed

Photoshop is not Fluid

Photoshop is not Fluid

- Largely Single Threaded

Photoshop is not Fluid

- Largely Single Threaded
  - Except Low Level Image Processing

Adobe

Photoshop is not Fluid

- Largely Single Threaded
  - Except Low Level Image Processing
  - User Input Interleaved with Rendering

Put Photoshop on a Separate Thread

Put Photoshop on a Separate Thread

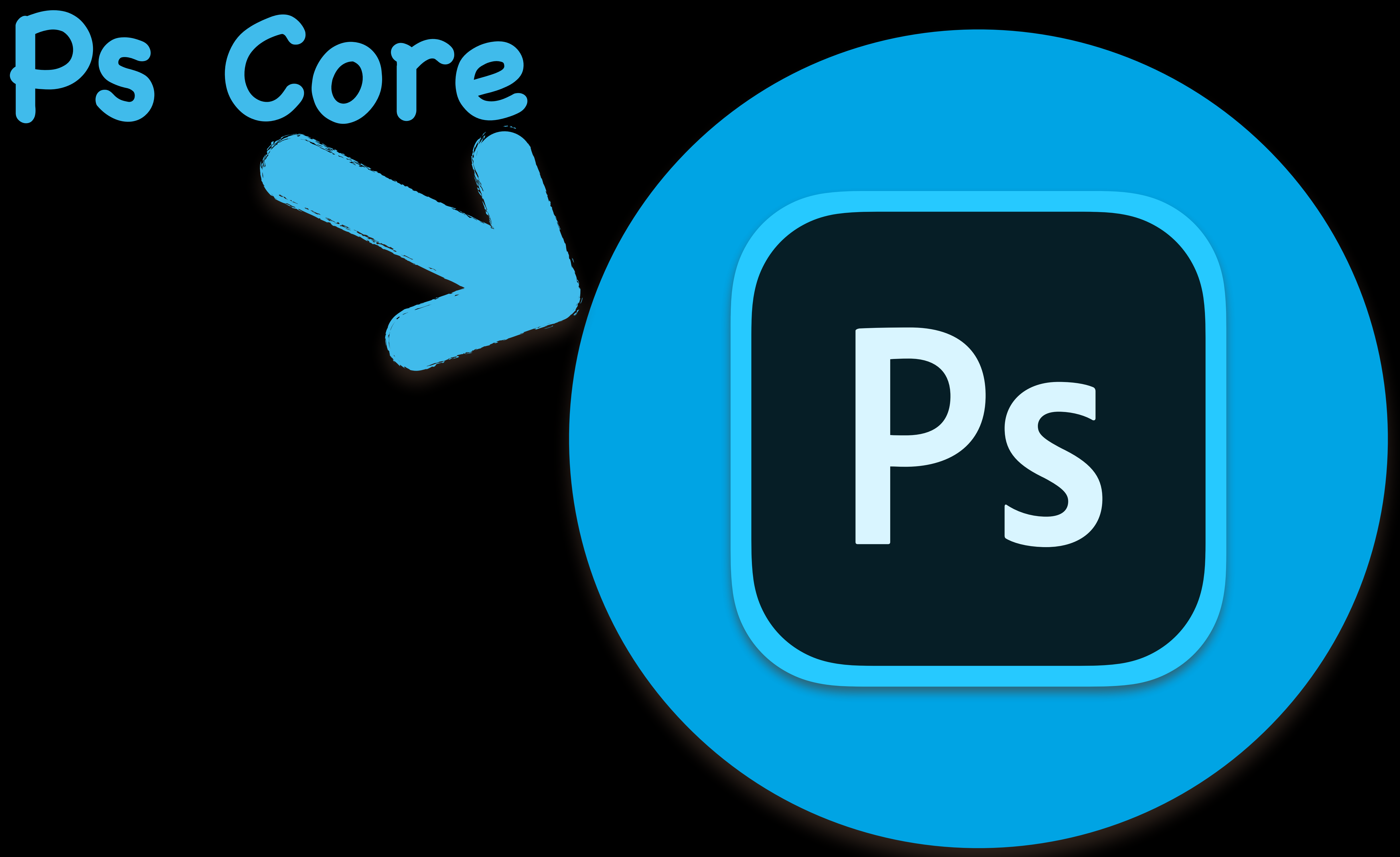
- Free UI thread to be Responsive

Put Photoshop on a Separate Thread

- Free UI thread to be Responsive
  - Runs independently, without blocking

Put Photoshop on a Separate Thread

- Free UI thread to be Responsive
  - Runs independently, without blocking



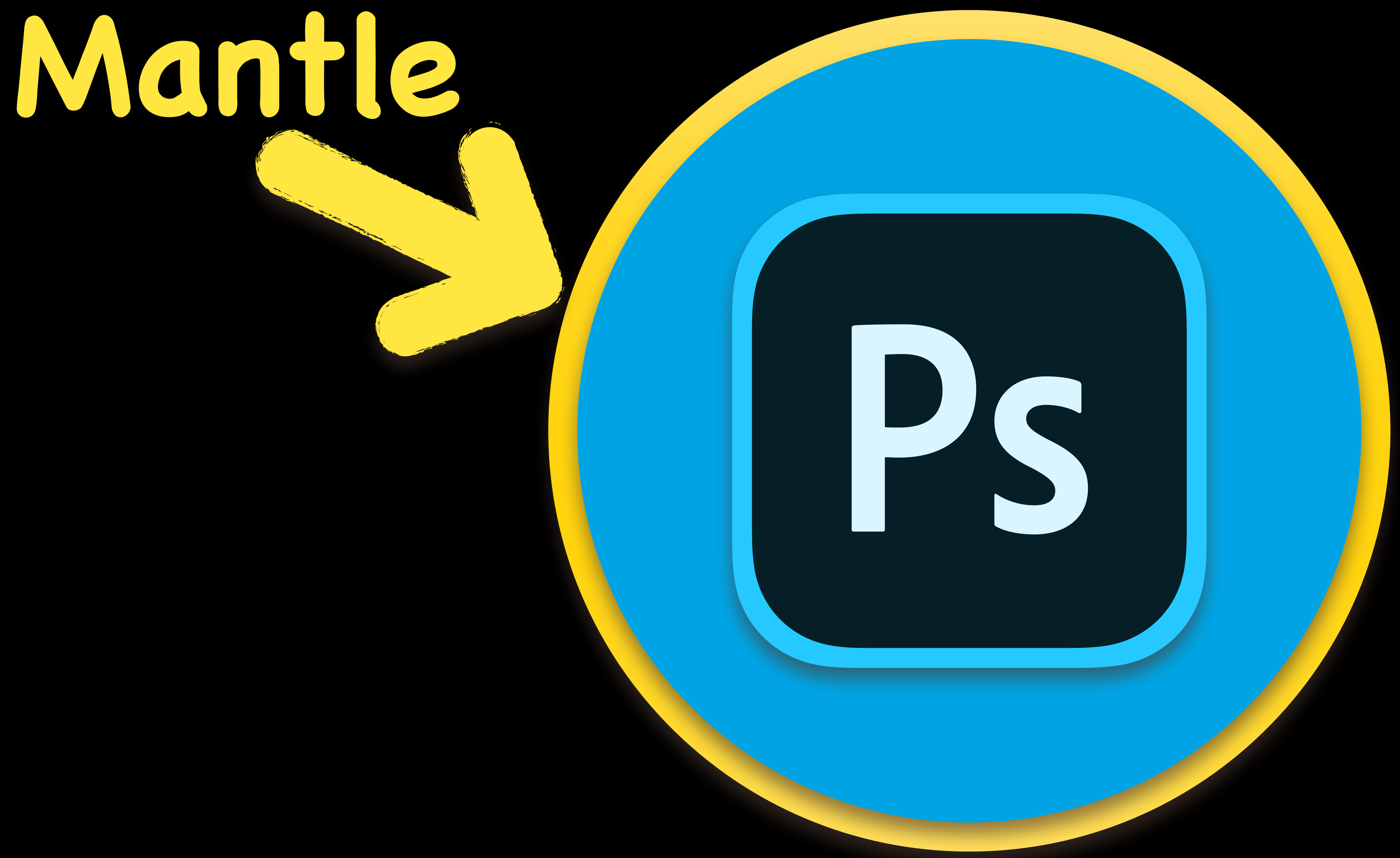
Mantle

Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application

Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application



Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application

Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application
  - The model is a containment hierarchy

Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application
  - The model is a containment hierarchy
    - `application` contains a collection of `document`s

Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application
  - The model is a containment hierarchy
    - `application` contains a collection of `document`s
    - `document` contains a collection of `layer`s, etc.

Adobe

Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application
  - The model is a containment hierarchy
    - `application` contains a collection of `document`s
    - `document` contains a collection of `layer`s, etc.
  - Properties present a rich interface

Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application
  - The model is a containment hierarchy
    - `application` contains a collection of `document`s
    - `document` contains a collection of `layer`s, etc.
  - Properties present a rich interface
    - read-only

Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application
  - The model is a containment hierarchy
    - `application` contains a collection of `document`s
    - `document` contains a collection of `layer`s, etc.
  - Properties present a rich interface
    - read-only
    - disableable (dynamic read-only)

Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application
  - The model is a containment hierarchy
    - `application` contains a collection of `document`s
    - `document` contains a collection of `layer`s, etc.
  - Properties present a rich interface
    - read-only
    - disableable (dynamic read-only)
    - connectable (slot & signal)

Mantle

- The *Mantle* is an API layer that presents an observable, shadow, model of the application
  - The model is a containment hierarchy
    - `application` contains a collection of `document`s
    - `document` contains a collection of `layer`s, etc.
  - Properties present a rich interface
    - read-only
    - disableable (dynamic read-only)
    - connectable (slot & signal)
    - auto disconnect on object expiration

Mantle

Mantle

- The Mantle manages high speed communication with Core

Adobe

Mantle

- The Mantle manages high speed communication with Core
  - Communication is handled by “sending code”

Mantle

- The Mantle manages high speed communication with Core
  - Communication is handled by “sending code”
    - No serialization/deserialization overhead

Mantle

- The Mantle manages high speed communication with Core
  - Communication is handled by “sending code”
    - No serialization/deserialization overhead

```
core_invoke(
    [file_type](TImageDocument* core_document) {
        core_document->SaveDocument(cSave, cSave, file_type);
    },
    transaction, core_document);
```

Mantle

- The Mantle manages high speed communication with Core
  - Communication is handled by "sending code"
    - No serialization/deserialization overhead

```
core_invoke(
    [file_type](TImageDocument* core_document) {
        core_document->SaveDocument(cSave, cSave, file_type);
    },
    transaction, core_document);
```

Mantle

- The Mantle manages high speed communication with Core
  - Communication is handled by “sending code”
    - No serialization/deserialization overhead

```
core_invoke(
    [file_type](TImageDocument* core_document) {
        core_document->SaveDocument(cSave, cSave, file_type);
    },
    transaction, core_document);
```

Mantle

- The Mantle manages high speed communication with Core
  - Communication is handled by “sending code”
    - No serialization/deserialization overhead

```
core_invoke(
    [file_type](TImageDocument* core_document) {
        core_document->SaveDocument(cSave, cSave, file_type);
    },
    transaction, core_document);
```

Mantle

- The Mantle manages high speed communication with Core
  - Communication is handled by “sending code”
    - No serialization/deserialization overhead

```
core_invoke(
    [file_type](TImageDocument* core_document) {
        core_document->SaveDocument(cSave, cSave, file_type);
    },
    transaction, core_document);
```

Mantle

- The Mantle manages high speed communication with Core
  - Communication is handled by "sending code"
    - No serialization/deserialization overhead

```
core_invoke(
    [file_type](TImageDocument* core_document) {
        core_document->SaveDocument(cSave, cSave, file_type);
    },
    transaction, core_document);
```

Mantle

- The Mantle manages high speed communication with Core
  - Communication is handled by “sending code”
    - No serialization/deserialization overhead

```
core_invoke(
    [file_type](TImageDocument* core_document) {
        core_document->SaveDocument(cSave, cSave, file_type);
    },
    transaction, core_document);
```

Transactions

Transactions

- Transaction system allows simple speculative execution

Transactions

- Transaction system allows simple speculative execution
  - Self correcting

Transactions

- Transaction system allows simple speculative execution
  - Self correcting
- Similar model to Apple's CoreAnimation

Transactions

- Transaction system allows simple speculative execution
  - Self correcting
- Similar model to Apple's CoreAnimation
  - Programmer's view (and user view) is instantaneous

Transactions

- Transaction system allows simple speculative execution
  - Self correcting
- Similar model to Apple's CoreAnimation
  - Programmer's view (and user view) is instantaneous
  - Even with underlying latency

Transactions

Transactions

- When a property is changed, the change is reflected immediately

Transactions

- When a property is changed, the change is reflected immediately
  - And the property is stamped with a transaction count

Transactions

- When a property is changed, the change is reflected immediately
  - And the property is stamped with a transaction count
- Each property change message from mantle to core is associated the transaction count

Transactions

- When a property is changed, the change is reflected immediately
  - And the property is stamped with a transaction count
- Each property change message from mantle to core is associated the transaction count
- The transaction id is stored in a thread-local scope

Transactions

- When a property is changed, the change is reflected immediately
  - And the property is stamped with a transaction count
- Each property change message from mantle to core is associated the transaction count
- The transaction id is stored in a thread-local scope
  - Globally available to any notifiers

Transactions

- When a property is changed, the change is reflected immediately
  - And the property is stamped with a transaction count
- Each property change message from mantle to core is associated the transaction count
- The transaction id is stored in a thread-local scope
  - Globally available to any notifiers
- Notifications from core to mantle echo back the count

Transactions

- When a property is changed, the change is reflected immediately
  - And the property is stamped with a transaction count
- Each property change message from mantle to core is associated the transaction count
- The transaction id is stored in a thread-local scope
  - Globally available to any notifiers
- Notifications from core to mantle echo back the count
  - If a mantle property receives an update with a count less than the property count, the update is ignored

Transactions

```cpp
class transaction {
public:
    enum class id_t : std::size_t { initial = 0, none = std::numeric_limits<std::size_t>::max() };

    transaction(id_t id) : _prior{current()} { current() = id; }

    ~transaction() { current() = _prior; }

    static id_t next() {
        static std::atomic<std::size_t> id{0};
        return static_cast<id_t>(++id);
    }

    static id_t& current() {
        thread_local id_t id{id_t::none};
        return id;
    }

private:
    id_t _prior;
};
```

High Frequency Channels

High Frequency Channels

- A *high frequency* event is any event expected to occur faster than the events can be processed

High Frequency Channels

- A *high frequency* event is any event expected to occur faster than the events can be processed
  - Screen update requests

High Frequency Channels

- A *high frequency* event is any event expected to occur faster than the events can be processed
  - Screen update requests
  - Property values changed by a slider

High Frequency Channels

- A *high frequency* event is any event expected to occur faster than the events can be processed
  - Screen update requests
  - Property values changed by a slider
  - Painting

High Frequency Channels

- A *high frequency* event is any event expected to occur faster than the events can be processed
  - Screen update requests
  - Property values changed by a slider
  - Painting

- Upon an initial event a coroutine is created and sent to be processed on core

High Frequency Channels

- A *high frequency* event is any event expected to occur faster than the events can be processed
  - Screen update requests
  - Property values changed by a slider
  - Painting

- Upon an initial event a coroutine is created and sent to be processed on core
- A direct communication channel is established from sender to coroutine

High Frequency Channels

- A *high frequency* event is any event expected to occur faster than the events can be processed
    - Screen update requests
    - Property values changed by a slider
    - Painting

- Upon an initial event a coroutine is created and sent to be processed on core
- A direct communication channel is established from sender to coroutine
- Subsequent messages are sent direct to the coroutine

High Frequency Channels

High Frequency Channels

- The coroutine manages adapting to the performance requirements of the stream

High Frequency Channels

- The coroutine manages adapting to the performance requirements of the stream
  - Take *latest* value

High Frequency Channels

- The coroutine manages adapting to the performance requirements of the stream
  - Take *latest* value
  - Coalesce values

High Frequency Channels

- The coroutine manages adapting to the performance requirements of the stream
  - Take *latest* value
  - Coalesce values
  - Process in *preview* mode

High Frequency Channels

- The coroutine manages adapting to the performance requirements of the stream
  - Take *latest* value
  - Coalesce values
  - Process in *preview* mode
  - Cancel operations that are no longer necessary

High Frequency Channels

- The coroutine manages adapting to the performance requirements of the stream
  - Take *latest* value
  - Coalesce values
  - Process in *preview* mode
  - Cancel operations that are no longer necessary
  - Defer low-priority operations to completion

High Frequency Channels

- The coroutine manages adapting to the performance requirements of the stream
  - Take *latest* value
  - Coalesce values
  - Process in *preview* mode
  - Cancel operations that are no longer necessary
  - Defer low-priority operations to completion

- Strategy is time based to maintain frame rate

High Frequency Channels

- The coroutine manages adapting to the performance requirements of the stream
  - Take *latest* value
  - Coalesce values
  - Process in *preview* mode
  - Cancel operations that are no longer necessary
  - Defer low-priority operations to completion

- Strategy is time based to maintain frame rate
  - Not user event based (such as finger up/down)

```cpp
template <class, class, class>
class latest;

template <class Executor, class Task, class... Args>
class latest<Executor, Task, void(Args...)> {

    struct receiver {
        Task _task;
        std::mutex _mutex;
        bool _queued{false}; // is a task queued already
        std::optional<std::tuple<Args...>> _value; // message value

        template <class F>
        explicit receiver(F&& f) : _task(std::forward<F>(f)) {}

        template <class... T>
        bool send(T&&... arg) {
            bool queued = true;
            std::unique_lock<std::mutex> lock(_mutex);
            _value = std::make_tuple(std::forward<T>(arg)...);
            std::swap(queued, _queued);
            return queued;
        }

        void invoke() {
            std::tuple<Args...> value;
            {
                std::unique_lock<std::mutex> lock(_mutex);
                _queued = false;
```

```
struct receiver {
    Task _task;
    std::mutex _mutex;
    bool _queued{false}; // is a task queued already
    std::optional<std::tuple<Args...>> _value; // message value

    template <class F>
    explicit receiver(F&& f) : _task(std::forward<F>(f)) {}

    template <class... T>
    bool send(T&&... arg) {
        bool queued = true;
        std::unique_lock<std::mutex> lock(_mutex);
        _value = std::make_tuple(std::forward<T>(arg)...);
        std::swap(queued, _queued);
        return queued;
    }

    void invoke() {
        std::tuple<Args...> value;
        {
            std::unique_lock<std::mutex> lock(_mutex);
            _queued = false;
            value = std::move(_value.get());
        }
        std::apply(_task, std::move(value));
    }
};

Executor _executor;
```

```
struct receiver {
    Task _task;
    std::mutex _mutex;
    bool _queued{false}; // is a task queued already
    std::optional<std::tuple<Args...>> _value; // message value

    template <class F>
    explicit receiver(F&& f) : _task(std::forward<F>(f)) {}

    template <class... T>
    bool send(T&&... arg) {
        bool queued = true;
        std::unique_lock<std::mutex> lock(_mutex);
        _value = std::make_tuple(std::forward<T>(arg)...);
        std::swap(queued, _queued);
        return queued;
    }

    void invoke() {
        std::tuple<Args...> value;
        {
            std::unique_lock<std::mutex> lock(_mutex);
            _queued = false;
            value = std::move(_value.get());
        }
        std::apply(_task, std::move(value));
    }
};

Executor _executor;
```

```cpp
struct receiver {
    Task _task;
    std::mutex _mutex;
    bool _queued{false}; // is a task queued already
    std::optional<std::tuple<Args...>> _value; // message value

    template <class F>
    explicit receiver(F&& f) : _task(std::forward<F>(f)) {}

    template <class... T>
    bool send(T&&... arg) {
        bool queued = true;
        std::unique_lock<std::mutex> lock(_mutex);
        _value = std::make_tuple(std::forward<T>(arg)...);
        std::swap(queued, _queued);
        return queued;
    }

    void invoke() {
        std::tuple<Args...> value;
        {
            std::unique_lock<std::mutex> lock(_mutex);
            _queued = false;
            value = std::move(_value.get());
        }
        std::apply(_task, std::move(value));
    }
};

Executor _executor;
```

```cpp
struct receiver {
    Task _task;
    std::mutex _mutex;
    bool _queued{false}; // is a task queued already
    std::optional<std::tuple<Args...>> _value; // message value

    template <class F>
    explicit receiver(F&& f) : _task(std::forward<F>(f)) {}

    template <class... T>
    bool send(T&&... arg) {
        bool queued = true;
        std::unique_lock<std::mutex> lock(_mutex);
        _value = std::make_tuple(std::forward<T>(arg)...);
        std::swap(queued, _queued);
        return queued;
    }

    void invoke() {
        std::tuple<Args...> value;
        {
            std::unique_lock<std::mutex> lock(_mutex);
            _queued = false;
            value = std::move(_value.get());
        }
        std::apply(_task, std::move(value));
    }
};

Executor _executor;
```

```
struct receiver {
    Task _task;
    std::mutex _mutex;
    bool _queued{false}; // is a task queued already
    std::optional<std::tuple<Args...>> _value; // message value

    template <class F>
    explicit receiver(F&& f) : _task(std::forward<F>(f)) {}

    template <class... T>
    bool send(T&&... arg) {
        bool queued = true;
        std::unique_lock<std::mutex> lock(_mutex);
        _value = std::make_tuple(std::forward<T>(arg)...);
        std::swap(queued, _queued);
        return queued;
    }

    void invoke() {
        std::tuple<Args...> value;
        {
            std::unique_lock<std::mutex> lock(_mutex);
            _queued = false;
            value = std::move(_value.get());
        }
        std::apply(_task, std::move(value));
    }
};

Executor _executor;
```

```cpp
    };

    Executor _executor;
    std::shared_ptr<receiver> _shared;

public:
    template <class T, class F>
    latest(T&& executor, F&& task)
        : _executor{std::forward<T>(executor)}, _shared{std::make_shared<receiver>(
                                                    std::forward<F>(task))} {}

    template <class... T>
    void operator()(T&&... arg) const {
        if (!_shared->send(std::forward<T>(arg)...)) {
            _executor([_shared = _shared] { _shared->invoke(); });
        }
    }
};

template <class T, class Executor, class Task>
inline auto make_latest(Executor&& executor, Task&& task) {
    return latest<Executor, Task, T>{std::forward<Executor>(executor), std::forward<Task>(task)};
}
```

```cpp
    };

    Executor _executor;
    std::shared_ptr<receiver> _shared;

public:
    template <class T, class F>
    latest(T&& executor, F&& task)
        : _executor{std::forward<T>(executor)}, _shared{std::make_shared<receiver>(
                                                     std::forward<F>(task))} {}

    template <class... T>
    void operator()(T&&... arg) const {
        if (!_shared->send(std::forward<T>(arg)...)) {
            _executor([_shared = _shared] { _shared->invoke(); });
        }
    }
};

template <class T, class Executor, class Task>
inline auto make_latest(Executor&& executor, Task&& task) {
    return latest<Executor, Task, T>{std::forward<Executor>(executor), std::forward<Task>(task)};
}
```

Adobe

```
    };

    Executor _executor;
    std::shared_ptr<receiver> _shared;

public:
    template <class T, class F>
    latest(T&& executor, F&& task)
        : _executor{std::forward<T>(executor)}, _shared{std::make_shared<receiver>(
                                                     std::forward<F>(task))} {}

    template <class... T>
    void operator()(T&&... arg) const {
        if (!_shared->send(std::forward<T>(arg)...)) {
            _executor([_shared = _shared] { _shared->invoke(); });
        }
    }
};

template <class T, class Executor, class Task>
inline auto make_latest(Executor&& executor, Task&& task) {
    return latest<Executor, Task, T>{std::forward<Executor>(executor), std::forward<Task>(task)};
}
```

```cpp
    };

    Executor _executor;
    std::shared_ptr<receiver> _shared;

public:
    template <class T, class F>
    latest(T&& executor, F&& task)
        : _executor{std::forward<T>(executor)}, _shared{std::make_shared<receiver>(
                                                    std::forward<F>(task))} {}

    template <class... T>
    void operator()(T&&... arg) const {
        if (!_shared->send(std::forward<T>(arg)...)) {
            _executor([_shared = _shared] { _shared->invoke(); });
        }
    }
};

template <class T, class Executor, class Task>
inline auto make_latest(Executor&& executor, Task&& task) {
    return latest<Executor, Task, T>{std::forward<Executor>(executor), std::forward<Task>(task)};
}
```

High Frequency Channels

```
auto f = make_latest<void(int)>(serial_queue, [](int x) { cout << x << '\n'; });

for (int n = 0; n != 3000; ++n) {
    f(n);
}
```

High Frequency Channels

```
auto f = make_latest<void(int)>(serial_queue, [](int x) { cout << x << '\n'; });

for (int n = 0; n != 3000; ++n) {
    f(n);
}
```

```
470
1066
1143
1347
1454
1507
1698
1930
2219
2649
2999
```

High Frequency Channels

```
auto f = make_latest<void(int)>(serial_queue, [](int x) { cout << x << '\n'; });

for (int n = 0; n != 3000; ++n) {
    f(n);
}
```

```
470
1066
1143
1347
1454
1507
1698
1930
2219
2649
2999
```

Mantle

Mantle

- All communication with client on client thread

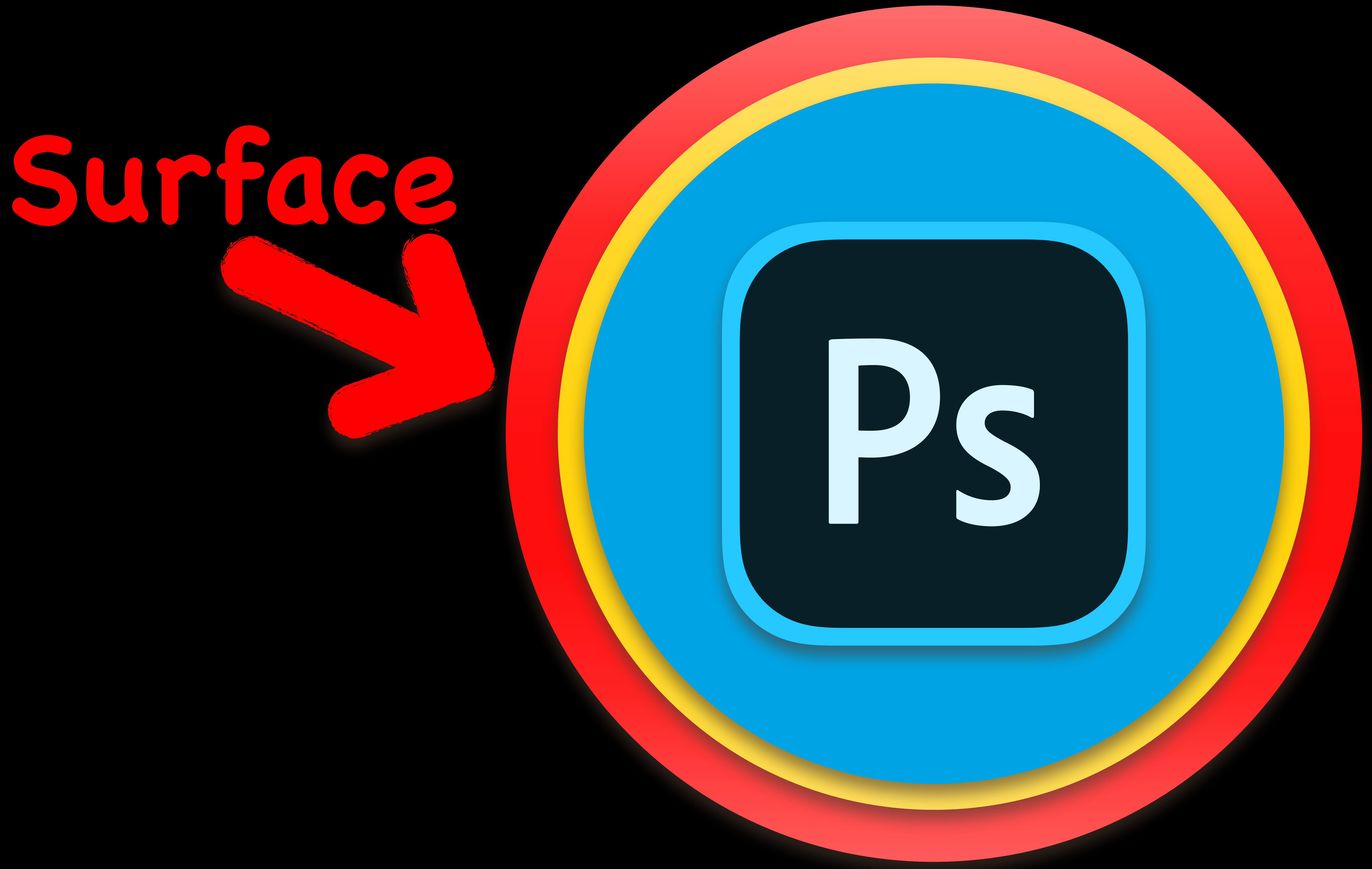
Mantle

- All communication with client on client thread
- The core and mantle are packaged as a library

Surface

Surface

- The *Surface* is a new, thin, UI



Surface

- The *Surface* is a new, thin, UI
  - Binds to the Mantle

Surface

- The *Surface* is a new, thin, UI
  - Binds to the Mantle
  - Model behaviors driven by the Core

Display System

Display System

- Canvas display consists of two layer

Display System

- Canvas display consists of two layer
  - Background is screen resolution of entire document

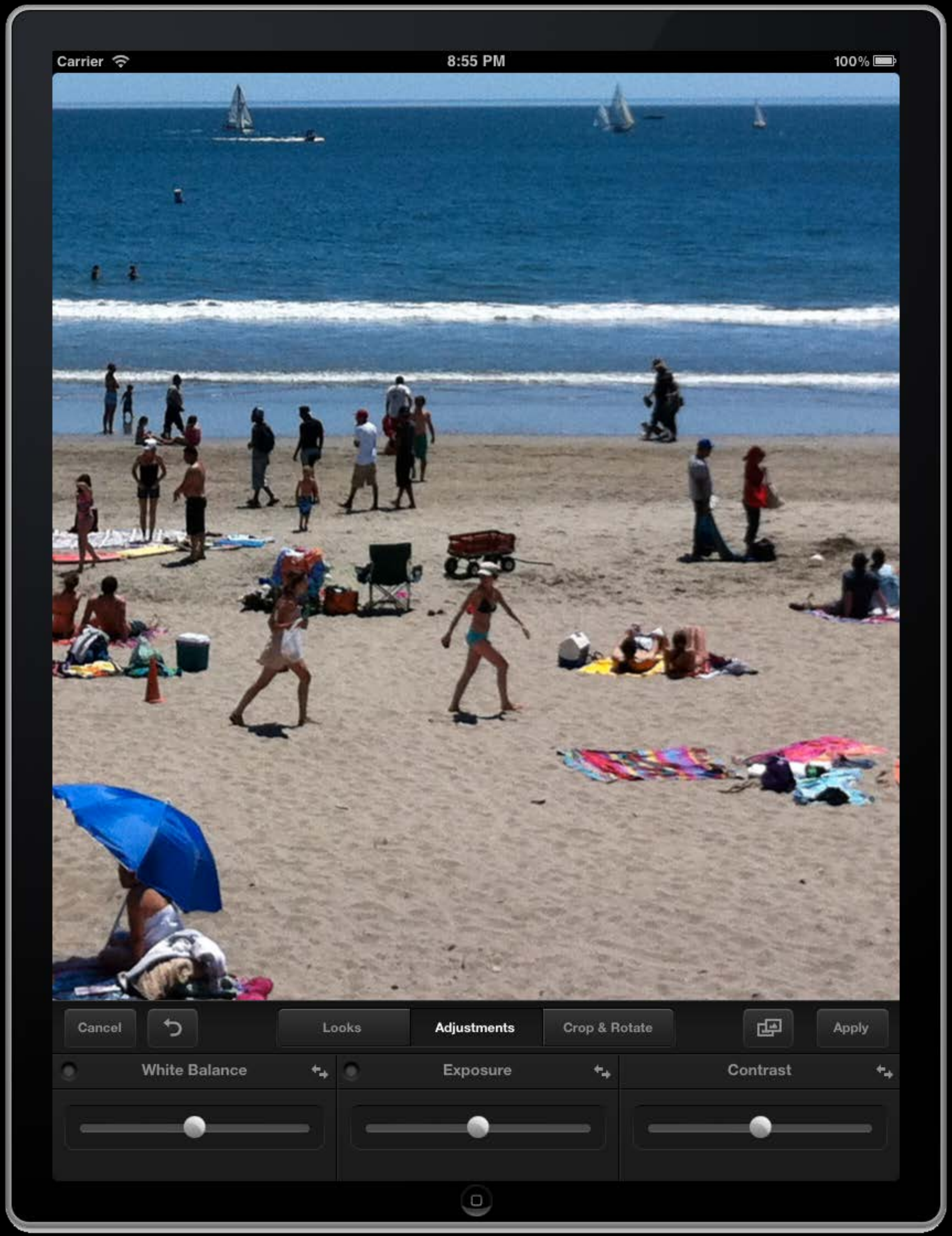
Display System

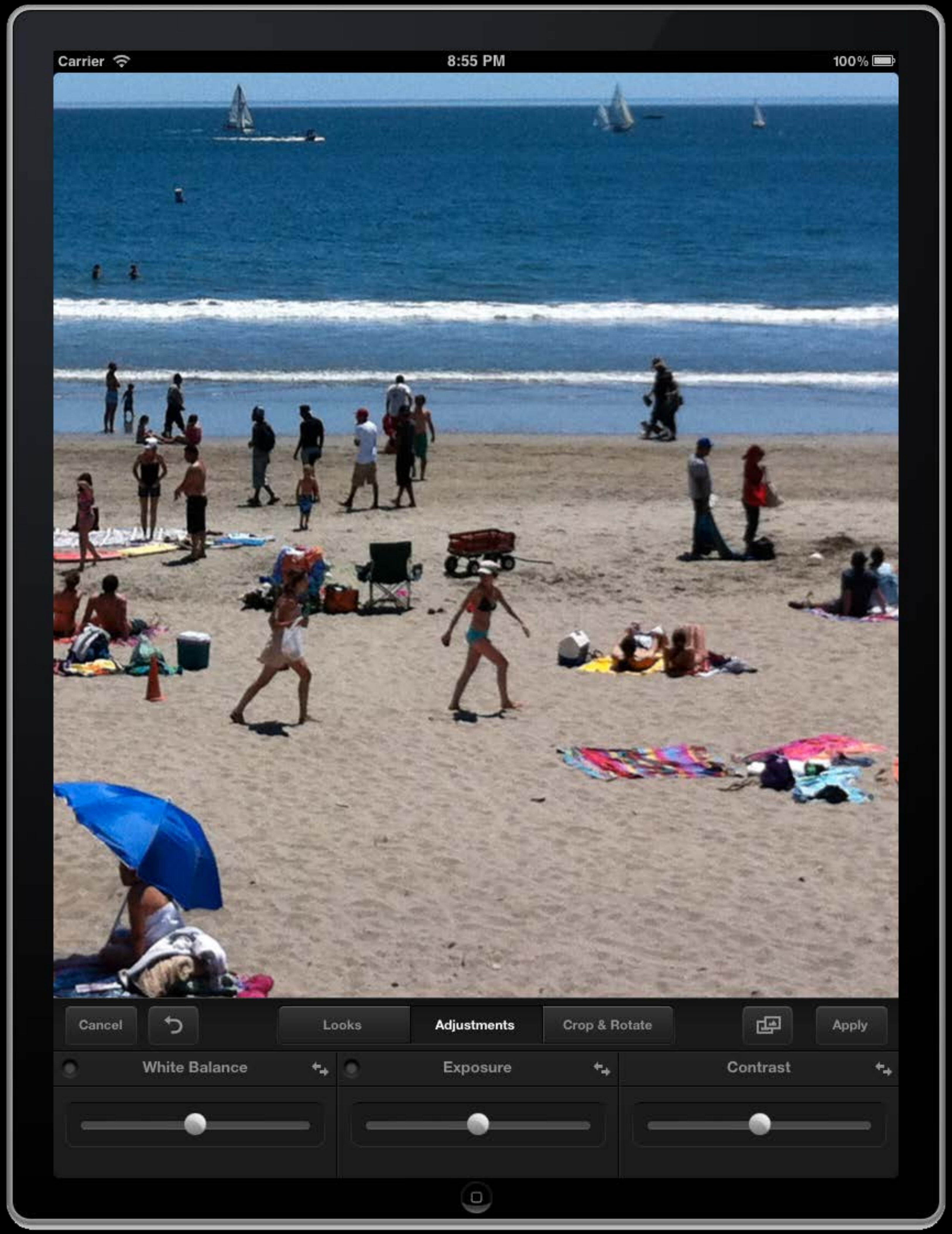
- Canvas display consists of two layer
  - Background is screen resolution of entire document
  - Foreground is screen resolution* of visible area

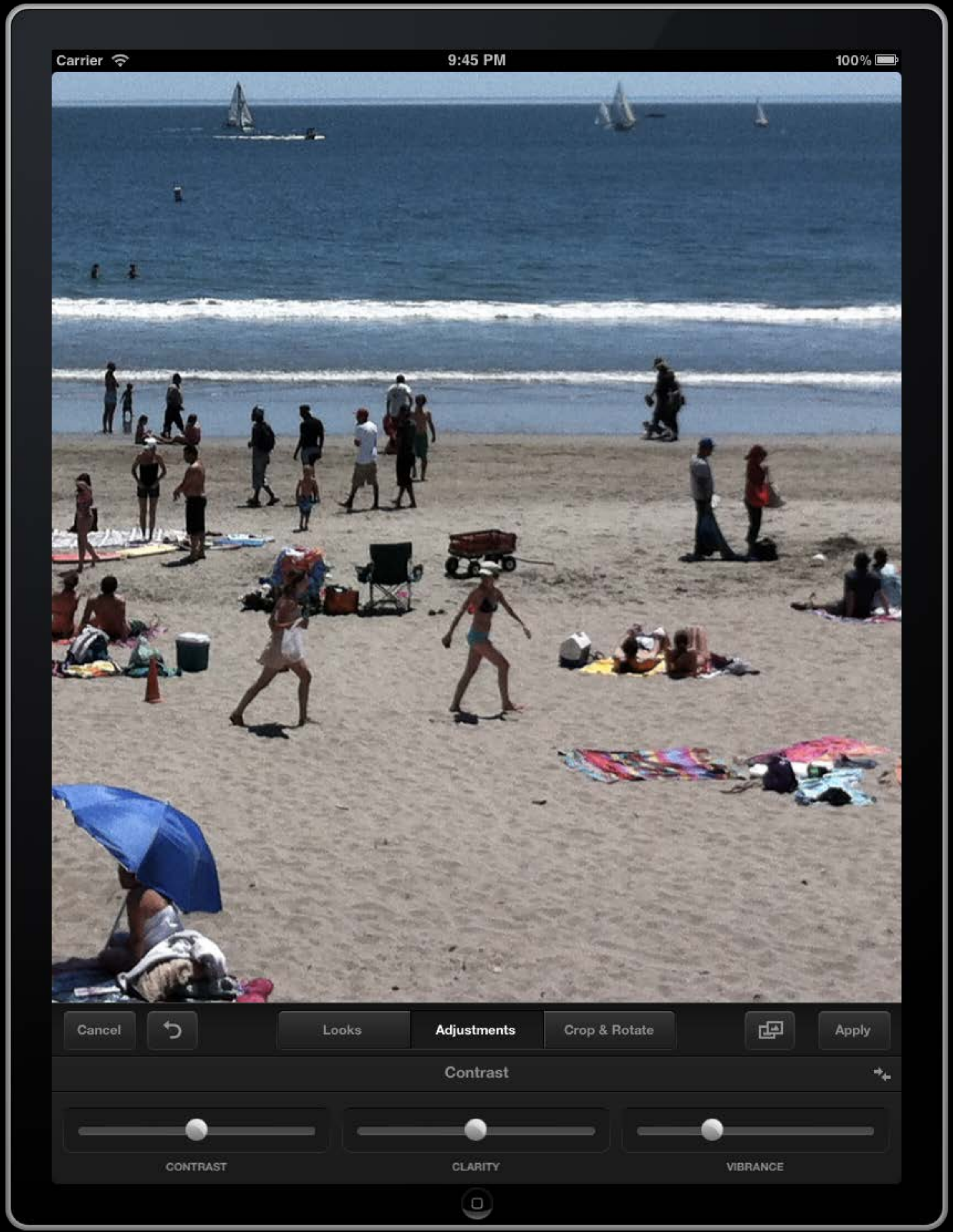
Display System

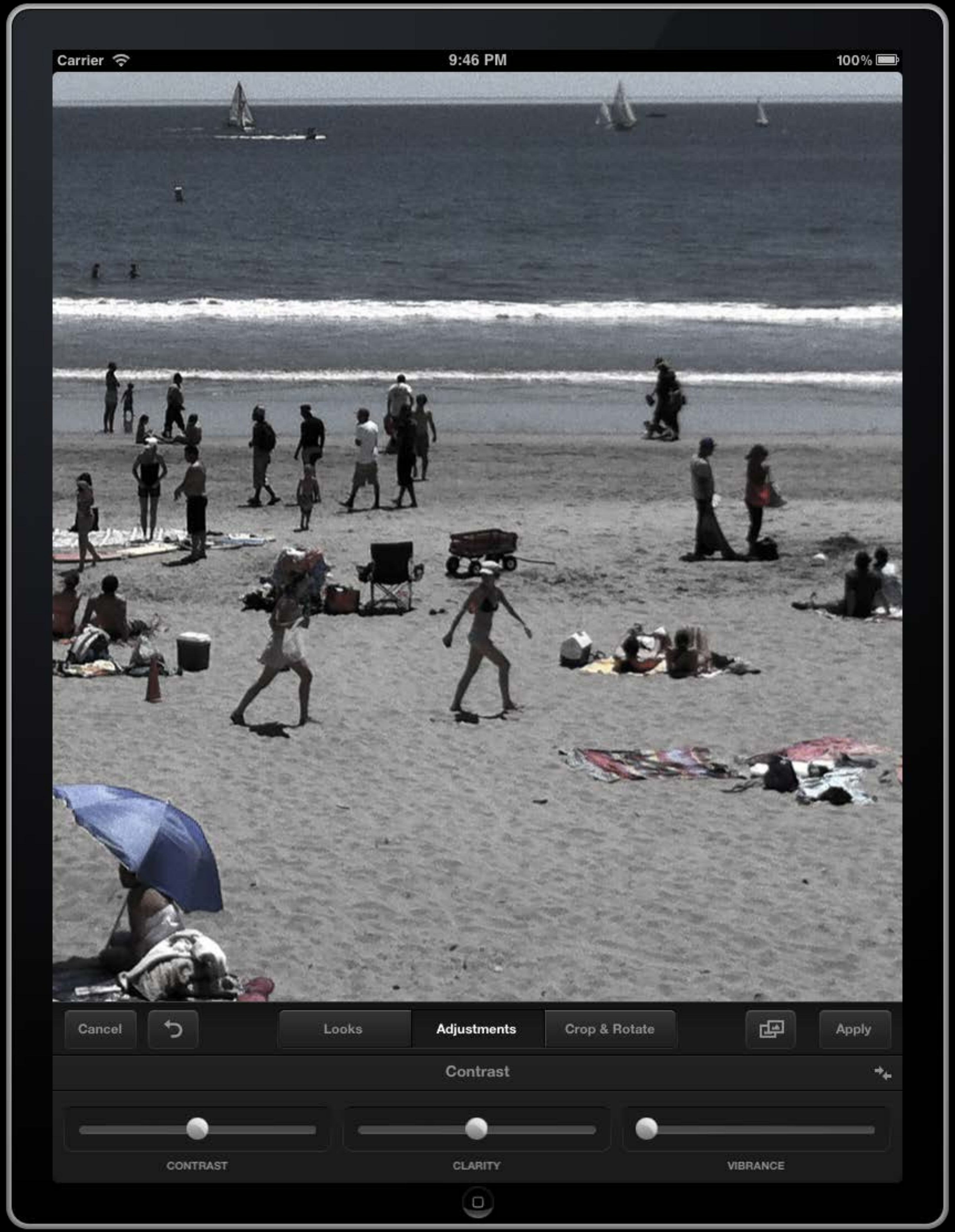
- Canvas display consists of two layer
  - Background is screen resolution of entire document
  - Foreground is screen resolution* of visible area

- *Resolution may be adapted to maintain frame rate

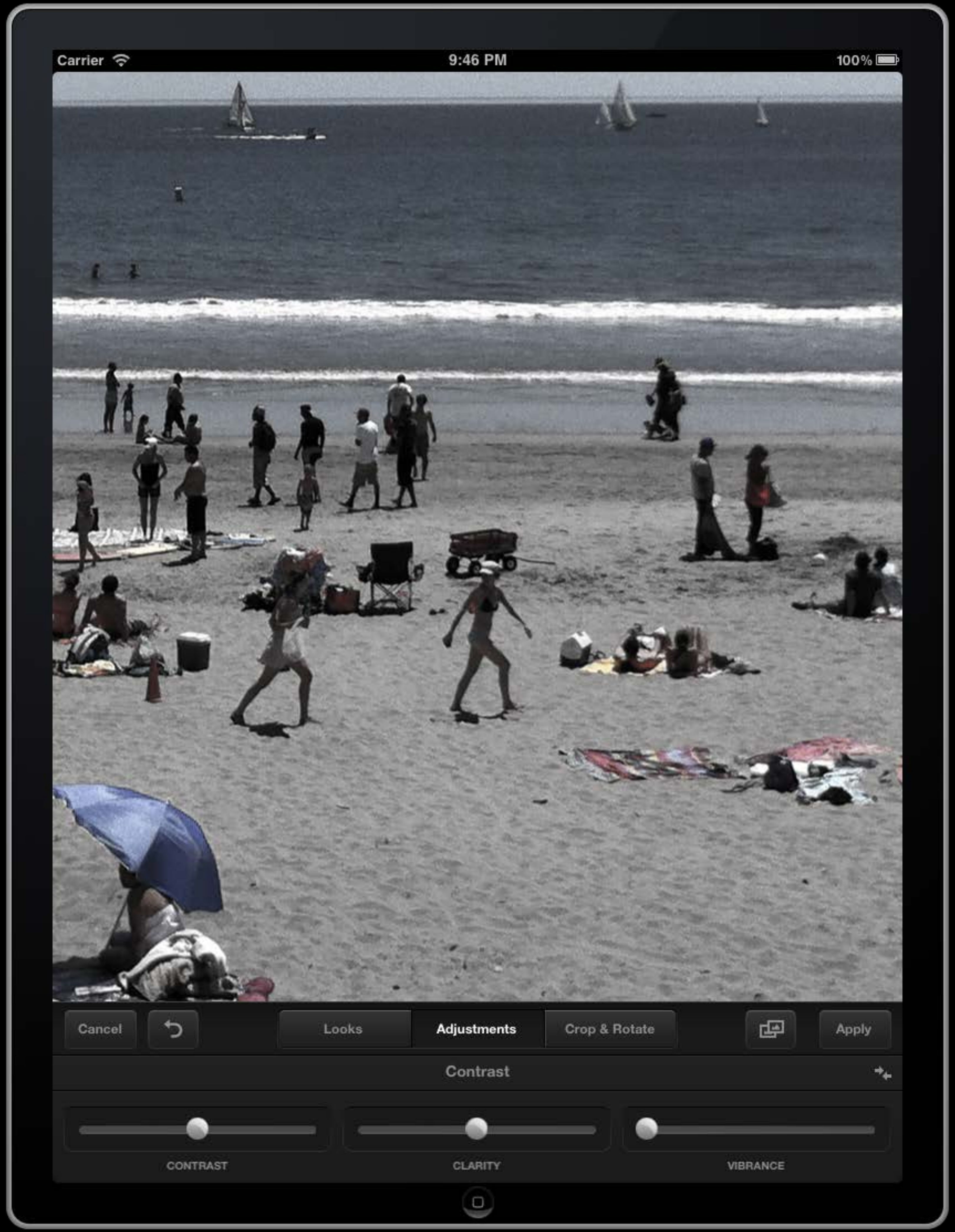

Carrier
8:55 PM
100%
Cancel
Looks
Adjustments
Crop & Rotate
Apply
White Balance
Exposure
Contrast


Carrier
8:55 PM
100%
Cancel
Looks
Adjustments
Crop & Rotate
Apply
White Balance
Exposure
Contrast


Carrier
9:45 PM
100%
Cancel
Looks
Adjustments
Crop & Rotate
Apply
Contrast
CONTRAST
CLARITY
VIBRANCE


Carrier
9:46 PM
100%
Cancel
Looks
Adjustments
Crop & Rotate
Apply
Contrast
CONTRAST
CLARITY
VIBRANCE

Adobe



Carrier
9:46 PM
100%
Cancel
Looks
Adjustments
Crop & Rotate
Apply
Contrast
CONTRAST
CLARITY
VIBRANCE


Carrier
10:01 PM
100%
Cancel
Looks
Adjustments
Crop & Rotate
Apply
Contrast
CONTRAST
CLARITY
VIBRANCE

Photoshop on the iPad

Adobe

Photoshop on the iPad

- Goal is to bring *real* photoshop to new devices and new customers
- Fast, fluid, and fully compatible with the desktop product

Photoshop on the iPad

- Goal is to bring *real* photoshop to new devices and new customers
- Fast, fluid, and fully compatible with the desktop product

- Enabled by just a few, simple, concepts

Photoshop on the iPad

- Goal is to bring *real* photoshop to new devices and new customers
- Fast, fluid, and fully compatible with the desktop product

- Enabled by just a few, simple, concepts

- **photoshop**is**hiring**.com

Adobe